MatPLC user manual

Foreword

This manual is available in several places and formats:

- On the web at http://mat.sourceforge.net/manual/ This version is updated every eight hours from the developers' version.
- In each distribution (alderaan, weekly, etc), in `doc/manual/index.html`
- As PDF or PostScript for printing; please ask on the mailing list for this, we do not have anything automated set up.

Table of Contents

- General
  - Introduction [p 4]
  - Who should read this manual [p 5]
  - System Requirements [p 7]
  - Safety Considerations
    Disclaimer of Warranty [p 9]
- Getting Started [p 11]
  - Usage scenarios [p 13]
    - PLC [p 14]
    - Traffic Cop [p 15]
    - Operator station [p 16]
    - SCADA server [p 17]
    - Intranet web server [p 18]
  - Demos [p 19]
    - Basic demo [p 20]
    - Basic_... demos [p 21]
    - Oven demos [p 23]
    - LPC demos [p 24]
- Program engineering [p 26]
  - Modularity [p 28]
  - Specifications [p 29]
  - HMI [p 30]
  - Testing [p 31]
  - Hints and Heuristics [p 32]
- Concepts
  - Modules [p 33]
  - Points [p 35]
- Config file [p 37]

    - PLC section [p 40]
    - Common configuration settings [p 45]
    - Configuration of synch [p 46]
    - Configuration of real-time features [p 47]
- Tools
    - matplc [p 50]
    - Config editor [p 51]
        - Module definition [p 53]
        - Point definition [p 54]
    - mattest / plctest [p 55]
- Modules
    - Logic engines
        - Classicladder [p 57] *(external project)*
        - Digital Signal Processing [p 58] (including PID control)
        - IEC 61131-3 ST and IL languages [p 68]
        - MAT IL language [p 72] (mnemonics)
        - PLC5 emulator [p 76]
        - ***planned:*** *OROCOS integration [p 77]*
    - HMI modules
        - HMI GTK (graphical) [p 78]
        - vitrine (text-based, output only) [p 88]
        - kbd (text-based, input only)
    - SCADA modules
        - Data logger [p 89]
        - hmiViewerServer [p 91] (visual)
    - Input/Output modules
        - Common I/O options [p 92]
        - 8255-based cards
        - abel [p 94] (AB PLC5 and similar, over Ethernet)
        - CIF [p 95] (ASi, CANopen, ControlNet, DeviceNet, Interbus, ModBus Plus, PROFIBUS, Sercos)
        - comedi I/O (various DAQ cards) [p 97]
        - Modbus [p 100]
        - Parallel port [p 106]
        - Pontech sv203b [p 108]
        - Simple UDP connection
- Tips and tricks
    - Bumpless transfers [p 109]
- Custom modules [p 110]
    - Languages other than C [p 112]
        - Python [p 113] *(perhaps the most similar style of language to BASIC and VB)*
        - Tcl [p 115]
    - Native C [p 117]

        - A basic custom module [p 118]
        - Example custom module [p 122]
        - General functions reference [p 126]
        - Global Map (gmm) reference [p 129]
        - Config (conffile) reference [p 135]
        - Logging (log) reference [p 138]
        - ...
        - Timer (timer, timer_ext) reference [p 140]
        - A custom I/O module [p 142]
- Contributing to the MatPLC [p 146]
    - Interfacing other projects to the MatPLC [p 148]
- Appendices
    - Glossary [p 151]
    - sample matplc.conf [p 153]
    - Log messages [p 164]
    - License terms [p 165]

$Date: 2005/08/20 06:11:42 $

[p 1] [p 1] [p 5]

Introduction

FIXME What is it? What can it be used for? (refer also to the usage scenarios [p 13] chapter)

Much like a real PLC, a particular MatPLC installation consists of several modules, plugged into a core (virtual backplane). There are I/O modules, logic modules, user interface modules etc. Most of this manual concerns the detailed instructions for one or another particular kind of module. Modules are futher explained in the Modules [p 33] chapter.

The central file in MatPLC is traditionally called `matplc.conf` (though you can give it any name you like). It determines what modules are running and any tuning parameters the modules require as well as configuring the MatPLC core, its global memory map and the like.

As such, the `matplc.conf` file is effectively the ‘main’ file in any installation: all options and settings are either in this file, or in some file it specifies. It is further described in the Config file [p 37] chapter

The Configuration Editor [p 51] will be the easiest way to create and edit `matplc.conf`, once it is finished. For now, you will still have to do most of the editing with a text editor.

[p 1] [p 1] [p 5]

$Date: 2004/12/28 05:31:58 $

[p 4] [p 1] [p 7]

Who should read this manual?

The MatPLC project has three or four faces it presents to the world, with corresponding roles that people play. We call these roles Operator, Application Builder, Module Builder and Core Builder.

At present, the bulk of this manual is aimed at the Application Builder, with some information for the Module Builder. There's not much of interest to the Operator at present, except perhaps out of curiosity.

Operator
: Interacts with the top level of the MatPLC, the HMI. This should be accessible to the average person on the factory floor.

Application Builder
: This can be an external integrator, or an in-house engineer, who does the end-user programming in languages such as stepladder, GRAFCET [p 57] and perhaps python [p 113] , or merely sets up things like PID loops [p 58] . This requires some skill, but it is largely also on the top level; it should be accessible, for instance, to an electrician with some experience of PLCs and PCs.

  The Application Builder should read or skim all of this manual with the exception of the Custom modules [p 110] chapter.

  For advanced applications with special requirements, the Application Builder will also have to be a bit of a Module Builder. The python scripting language [p 113] is somewhat on the borderline between the two levels.

Module Builder
: At the middle level is the API and programming interface, where modules are written. This should be accessible to the average programmer with fairly low effort.

  This is the level is where the bulk of the code of the MatPLC resides. This is the code that provides most of the functionality to support the Application Builders and End Users and generally makes the MatPLC a useful project.

  The Custom modules [p 110] chapter is specifically aimed at the Module Builder.

Core Builder
: Finally, there is the low level, where the somewhat cryptic core of the project is written.

Each of these roles also corresponds to a different way of contributing to the project - see the Contributing chapter [p 146] if you're interested in more.

Note: the phrasing "should be accessible" above reflects design, not always reality. If there’s some aspect of the level that is not accessible to the designated audience, it’s a bug; please report it using the buglist.

[p 4] [p 1] [p 7]

$Date: 2004/12/28 05:32:10 $

[p 5] [p 1] [p 9]

System Requirements

Because of the high modularity of the MatPLC, you do not necessarily need all the software listed below. No-one will ever use all the modules; if you do not need a module, you do not need its requirements.

Common requirements

As well as a basic Linux system, compiling and running the MatPLC system requires a few additional packages.

When we make a binary release, we will list here the packages required to run that release.

To compile and run the MatPLC, you'll need gcc, libtool, make and the other usual packages involved in compiling - header files, linkers, etc.

In addition, you will almost certainly need the *console-tools development* package. The basic demo depends on this, and it's usually not installed by default.

Note: the libraries that come with **Debian potato** are missing some of the functions that we use. One workaround is to update parts of the system to **Debian woody**, which is the new version of Debian - download the new `libc6-dev` and whatever it depends on (let `apt` or `dselect` do it for you).

Perl is used in various places in the project; it's perhaps not strictly necessary, but it'll be a hassle going without.

Module requirements

These are the requirements of the individual modules.

doc
: To convert the manual to ps and pdf formats, you will need html2ps, ghostscript, ps2pdf and preferably also the LaTeX hyphenation files. The lca-2002 talk requires full LaTeX and netpbm, but it can be downloaded off the webpage instead.

logic/iec
: flex, bison

mmi/curses
: curses

mmi/hmi_gtk
: glade (with gnome support) development, gtkextra library; the `widgetnamer`wizard also requires python (with gtk)

linuxkbd
: console-tools development (note that this module is used by the basic demo)

io/comedi
: comedi library

Summary

This section will contain the lists of packages for various distributions, corresponding to the above requirements lists, which can simply be fed to the appropriate installation tool (for instance, to Debian's `apt-get install`).

At present, it's rather incomplete.

Debian
: `console-tools-dev libcomedi-dev libgnome-dev libglade-gnome0-dev libgtkextra-dev`

[p 5] [p 1] [p 9]

$Date: 2004/12/28 05:31:58 $

[p 7] [p 1] [p 11]

Safety

Safety of the system should be top priority of any design. Even the control loop of a model railway can become dangerous if mains voltage finds its way from the control system to the model’s ground plane.

Safety should be taken into account from the earliest stage of design. Safe design is always more complex, more difficult to design and more costly to implement than plain vanila safety-ignorant one - but it is much easier and cheaper if safety aspects are taken into account from the very beginning.

Even where lives or injuries are not at stake, fail-safe design is always preferrable (you do not want to have your garden watering system stay ON indefinitely if the RCD breaker drops mains during watering...)

MAT is currently still in development phase and as such it is not suitable for safety sensitive applications.

Responsibility for Safety

Any system should be designed to be failure-safe if failure may cause serious damage or injury. Final responsibility for design - including choice of hardware and software - always rests with actual designer, implementor and user of the whole system.

The designer, implementor and user should also make sure they comply with any relevant statutory and common law requirements with regards to safety, licensing, registration and permits and the like.

Since you did not pay us (the developers of MatPLC), we cannot provide any warranty or any other sort of statement as to the actual performance of the software, its documentation or any other part.

Formal Disclaimer of Warranty

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED BY THE LICENCE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL

DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Obtaining warranty

It is permissible for third parties to offer warranty protection for this software in exchange for a fee, or to offer warranty protection for a particular system which uses MatPLC for a fee, and so on. This is always up to the parties involved, and should be in writing to avoid misunderstandings and unnecessary costly litigation.

If you offer such warranties, please send us an e-mail so that we can list you on our webpage.

[p 7] [p 1] [p 11]

$Date: 2004/12/28 05:31:58 $

[p 9] [p 1] [p 13]

Getting started with the MatPLC

This page will explain how to find one’s way around the MatPLC. It is rather brief at the moment.

Running the basic demo

The demo is a very primitive four-light chaser. Its only purpose is to make sure that the MatPLC downloaded and compiled OK.

After downloading the tarball and unpacking it in a directory, change to the `demo/basic` subdirectory and run `make`. Once it compiles everything, run the `demo` script in that directory.

During compilation, a message might come up saying that `linuxkbd` could not be compiled. This is merely a warning; the demo can run without it. Just press Enter to continue - an alternate keyboard interface will be used.

Once the demo is running, you’ll see a star moving across the middle of the screen. The direction can be changed using **L** and **R** (which may act as toggle buttons, if `linuxkbd` didn’t compile). To exit, hit **Q**.

If you have trouble starting up the MatPLC, try running
`../../tools/run/matplc -s`
just in case there’s a half-started MatPLC around. The `-s` option stands for "shutdown", and makes sure the slate is clean. Then try running the `demo` script again.

Other than that, please ask on the mailing list, giving as much information as you can about how far you got and what failed.

Running the GUI-based oven demo

You will need to be in X for this demo to run (the default on modern Linux boxes). Change to the `demo/oven_gkt` subdirectory, run `make` and then the `demo` script in that directory.

To see the demo in action, first set the machine to automatic and start it, then set the setpoints according the table below, or something similar to that:

FluidTempSP	200	°C
BeltSpeed	10	m/min
AirTempSP Zone1	100	°C
AirTempSP Zone2	120	°C
AirTempSP Zone3	140	°C

Creating your own project

This section not yet written. **FIXME**

[p 9] [p 1] [p 13]

$Date: 2004/12/28 05:31:58 $

[p 11] [p 1] [p 14]

Usage scenarios

The MatPLC comes with a plethora of modules and options, which can be somewhat confusing at first sight. The following sections will describe five envisaged usage scenarios for the MatPLC.

These are guides only, of course. They can be modified, or combined; depending on the situation, one computer may well fulfil all five of the roles described. In other cases, it would be wise to separate them, for instance for the sake of reliability.

[p 11] [p 1] [p 14]

$Date: 2004/12/28 05:32:12 $

[p 13] [p 1] [p 15]

PLC usage scenario

This scenario is perhaps the simplest one. Here, the PC controls (or monitors) a machine, much like a PLC might.

Example of simple hardware configuration: PC with inserted relay I/O card, switching on lights and taking photos of some process everytime something interesting happens (e.g. lab mouse comes to take food).

Usually, it would consist of a single logic module (for instance the IL language [p 72]) and a single I/O module.

Alternately, one might use the python scripting language [p 113] instead of a logic module, thus getting roughly the equivalent of a VB+OPC environment.

Optional additions are some sort of communication, for instance Modbus or one of the CIF [p 95] cards, and the DSP module [p 58] for PID and general floating-point operations.

[p 13] [p 1] [p 15]

$Date: 2004/12/28 05:32:13 $

[p 14] [p 1] [p 16]

Traffic Cop usage scenario

In this scenario, the PC will be the hub between two or more disparate industrial busses or networks, copying data between them pretty much without change and without any actual logic or intelligence.

This scenario contains no logic module, only two (or more) bus or network communications modules. The mapping between the various modules is specified in the `matplc.conf` file itself.

In the config, there will be a series of clauses looking pretty much like this, one for each point:

```
PLC:point L1 "light 1" bus1
bus1:map in bit.1 L1
bus2:map out bit.1 L1
```

A point can also be mapped to more than one output, like this:

```
PLC:point L1 "light 1" bus1
bus1:map in bit.1 L1
bus2:map out bit.1 L1
bus3:map out bit.42 L1
```

An addition to this would be to have one of the logic modules to do some massaging of the data. For instance, the DSP module [p 58] could convert between units:

```
PLC:point distIN "distance in inches" bus1
PLC:point distMM "distance in mm" dsp
bus1:map in register.1 distIN
dsp:fblock add distMM distIN 25.4
bus2:map out register.1 distMM
```

[p 14] [p 1] [p 16]

$Date: 2004/12/28 05:32:13 $

[p 15] [p 1] [p 17]

Operator Station usage scenario

Here, the PC will act as a HMI, communicating with the operators, giving information on machine and production status and allowing them to control it in turn.

This scenario centers around a HMI module, such as the GTK one [p 78] . It will also need one of the bus or network communications modules to talk to the rest of the system, to the PLC or PC doing the actual control.

Alternately, it can be combined with the PLC usage scenario [p 14] for an all-in-one solution.

Again, other modules can be added to the pure Operator Station. For instance, the DSP module's [p 58] `ramp` block could be used to limit the speed and acceleration at which the operator can change values to within the limits of the machinery.

[p 15] [p 1] [p 17]

$Date: 2004/12/28 05:32:13 $

[p 16] [p 1] [p 18]

SCADA server usage scenario

This scenario is similar to the Operator Station [p 16] scenario, except that instead of interacting with the operator directly, it does so through a separate SCADA program, perhaps remote over the network.

[p 16] [p 1] [p 18]

$Date: 2004/12/28 05:32:13 $

[p 17] [p 1] [p 19]

Intranet web server usage scenario

This scenario is similar to the Operator Station [p 16] scenario, except that instead of interacting with the operator directly, it does so through a web server. The operator then connects using any ordinary web browser.

While this is not really suitable for control, it is ideal for monitoring and reporting that is to be accessible throughout the corporate LAN.

The MatPLC interfaces with the Zope application server environment using the python language interface [p 113] ; it can also log data to MySQL database, the "M" in "LAMP", which can then be summarised on the web in the normal way. For an example of what can be done with this, see Juan’s on-line demo, which uses exactly this setup (MatPLC, Zope, MySQL and so on).

[p 17] [p 1] [p 19]

$Date: 2004/12/28 05:32:13 $

[p 18] [p 1] [p 20]

Introduction to demos

The MatPLC comes with a collection of demos, illustrating its capabilities and forming a sort of documentation-by-example. They are described in turn in this section.

`basic`
: Basic light chaser demo - does the code run at all? [p 20]

`basic_dsp`
`basic_dsp_gtk`
`basic_il`
`basic_modbus`
`basic_cif`
`basic_parport`
`basic_plc5`
`basic_synch`
`basic_tcl`
: Very simple demos demonstrating individual modules or features. [p 21]

`oven_gtk`
`oven`
: oven controller with graphical or text interface [p 23]

`lpc/test`
`lpc/arguments`
`lpc/chaser`
`lpc/messages`
`lpc/emailer`
`lpc/echo_network`
`lpc/echo_serial`
`lpc/ladder_basic`
`lpc/ladder_chaser`
`lpc/matd`
`lpc/das08`
`lpc/java_hmi`
: LPC demos [p 24] (temporary location in the manual, during the transition)

$Date: 2004/12/28 05:32:09 $

[p 19] [p 1] [p 21]

Basic demo

The basic demo is very primitive. Its main purpose is to ensure that the MatPLC has downloaded and compiled more-or-less correctly, as outlined in the Getting Started section [p 11] . It also forms the basis of the basic_... demos [p 21] , most of which add one single feature to this demo.

`basic`
: basic light chaser demo

It’s a simple light-chasing demo, controlled from the keyboard. No special hardware required.

To start:

```
cd demo/basic; make
```

To quit:
: press **Q** on the keyboard

It’s controlled from the keyboard (module Kbd, `kbd.c`): the keys **L**, **R** and **Q** on the keyboard toggle the three points ’left’, ’right’ and ’quit’.

The Chaser module (`chaser.c` in the `demo/basic` directory) is a primitive ’light chasing’ program. The direction of movement can be changed using the ’left’ and ’right’ points; these are notionally push-buttons, but in this demo they are controlled by the **L** and **R** keys. The speed of the chaser can be configured with the "delay" setting in the "Chaser" section of matplc.conf

The demo terminates when the ’quit’ point comes on (keyboard **Q**).

[p 19] [p 1] [p 21]

$Date: 2004/12/28 05:32:09 $

[p 20] [p 1] [p 23]

Basic_... demos

The `basic_...` demos illustrate various modules or facilities of the MatPLC in the simplest possible manner. Most of them are simply the basic demo [p 20] with a single additional feature, for instance outputting the chasing lights to a Modbus slave, or showing them in graphics. A few of them are unrelated, such as the DSP demo which is a simple PID loop.

As well as illustrating the features, they serve as documentation by example: the `matplc.conf` files in these directories are generally copiously commented, and are thus well worth reading for the configuration details of the module or facility in question.

They need more description here, **FIXME**. Some of it can be copied from the individual README files.

`basic_dsp`
: basic PID loop

`basic_dsp_gtk`
: a PID loop with graphical interface

`basic_il`
: implements the light chaser with IL code

`basic_modbus`
: modbus based communications (light chaser)

`basic_comedi`
: basic demo for comedi-based DAQ cards. Analog input on device 0, subdevice 0, channel 0 (displayed on screen); analog output on device 0, subdevice 1, channel 0, physical range 0.0-4.096 (controlled by a hmi_gtk slider).

`basic_cif`
: uses Synergetics CIF drivers for multiple network types (light chaser)

`basic_parport`
: control via parallel port (light chaser)

`basic_plc5`
: a light chaser using the PLC-5 simulator

`basic_synch`
: synchronisation of the modules (light chaser)

`basic_tcl`
: a TCL based GUI

[p 20] [p 1] [p 23]

$Date: 2004/12/28 05:32:09 $

[p 21] [p 1] [p 24]

Oven demos

The oven controller demos. This is the second demo mentioned in the Getting Started section [p 11] .

`oven_gtk`
: an oven controller with a graphical interface

`oven`
: the oven controller with a text interface

To see the demo in action, first set the machine to automatic and start it, then set the setpoints according the table below, or something similar to that:

FluidTempSP	200	°C
BeltSpeed	10	m/min
AirTempSP Zone1	100	°C
AirTempSP Zone2	120	°C
AirTempSP Zone3	140	°C

More description needed here, including the keyboard interface for the `oven` demo from the README. **FIXME**

[p 21] [p 1] [p 24]

$Date: 2004/12/28 05:32:09 $

[p 23] [p 1] [p 26]

LPC demos

The LPC demos are temporarily collected here, during the transition, pending a more detailed description.

They originate in the LPC project, which is in the process of merging with the MatPLC.

`lpc/test`
: LPC library simple test program

`lpc/arguments`
: LPC library argument passing

`lpc/chaser`
: LPC version of the basic light chaser

`lpc/messages`
: LPC library based message passing

`lpc/emailer`
: LPC library to send email

`lpc/echo_network`
: LPC library for echoing network communications

`lpc/echo_serial`
: LPC library for echoing serial communications

`lpc/ladder_basic`
: LPC library for executing basic ladder logic engine

`lpc/ladder_chaser`
: LPC library for the chaser on the ladder logic engine

`lpc/matd`
: LPC library based network interfacee

`lpc/das08`
: LPC library control of DAS08 DAQ card

`lpc/java_hmi`
: LPC library based interface for HMI (not fully operational)

[p 23] [p 1] [p 26]

$Date: 2004/12/28 05:32:09 $

[p 24] [p 1] [p 28]

Program Engineering

Introduction

One of the more difficult areas of programming is the overall organization and planning of the program. Because of this, it is usually the most important part, the part that makes the difference between a project that's on-time and working and a project that's a protracted nightmare and never quite works right (if at all).

Faced with a large problem for the first time, the beginning programmer frequently attempts to attack it in the same way as a small problem - typically by sitting down and beginning to code. This is made worse by programming books and courses, which of necessity must use relatively short examples (a couple of pages at most) and exercises (perhaps a dozen man-hours for a major assignment). Unfortunately, using this approach for real-world-sized projects usually leads to programs which, if they work at all, work badly, and are difficult to maintain and modify in the future.

Large problems demand their own programming techniques.

In reality, it's probably a good idea to apply them always; after all, if it's a small project which really is very simple, it won't take long to document it. If the design and documentation take much time and effort, then it probably isn't such a simple project after all.

This chapter cannot hope to cover this area; it is much too wide. Also, programming as a discipline is still young and many of these techniques are poorly developed. However, we hope to provide at least a quick overview of some of the important and uncontroversial points.

First, though, a quote that perhaps best summarizes the ideal of all design:

> Perfection is attained not when there is no longer anything to add but when there is no longer anything to take away. *(Antoine de Saint-Exupéry)*

Safety

A reminder that if there is any way in which the software could (through action or inaction) cause a hazardous situation or harm a human, safety-critical software specialists need to be brought in, or such situations eliminated by altering the design of the machine. After all, the computer is in many ways a single component, and thus a single point of failure; expecting it to be solely responsible for safety is unrealistic. Where at all possible, hardware interlocks should be provided to prevent dangerous combinations.

If it is not possible to entirely eliminate such situations, the reliability requirements on the software will be well beyond the scope of this short introduction, and indeed beyond the ability of most commercial software houses and general-purpose programmers and software engineers. Specialized techniques, procedures and methods will need to be applied. Neither MatPLC nor Linux itself have been put through this process.

[p 24] [p 1] [p 28]

$Date: 2004/12/28 05:32:12 $

[p 26] [p 1] [p 29]

Modularity

The key to solving large problems well is *modularity*. This means dividing the problem into sections (or modules, components, blocks etc) and solving each section separately.

A program might be divided along the process flow (infeed, process 1, accumulation, process 2, outfeed, palletizer); by mode (manual, single-step, automatic, callibration); by levels (single motions, compound motions, actions, tactics, strategy); by purpose (control, monitoring, mmi, comms); or in some other way that makes sense for the project. It might well be divided by some combination of the above.

Often, a section is still too large to be solved in one piece, so it is divided into sub-sections in a similar way, and so on for the sub-sections.

The trick with modularity - and the reason why it’s so important - is to ensure that each section does a single, well-defined job. It should interact with other sections sparingly; a sub-section should only interact within the section, unless its purpose is specifically interaction (in which case it should do little else). Obviously, like all rules, this can be broken on occasion. But not often.

[p 26] [p 1] [p 29]

$Date: 2004/12/28 05:32:12 $

[p 28] [p 1] [p 30]

Specifications

Or, writing it all down.

Program specifications fulfil the same role as engineering drawings and circuit diagrams: they help organize things so that nothing gets overlooked or left out, and they help communicate to the other people involved about what’s going on; just as importantly, they serve later on down the line when maintenance needs to be done. So, during the design of the program, make sure you write it all down.

No-one would start cutting and welding, or wiring up anything beyond a throw-away mock-up, without the proper drawings being made and signed off on. Programming is no different.

Each section and sub-section should have a one-page description of what it does and how it relates to other sections and sub-sections. If necessary, it should then have a larger document describing details; but that one-page description should be written so that it’s enough for someone trying to get the overall idea of the system.

Write down what the morning startup (or shift handover) will look like to the operator. What about recovery from various kinds of snags and stops? Can the operator clear a blockage and resume automatic mode? How? What about going to lunch - will the operator log out, shut down the machine? (These are called "use cases".)

Talk to the other people involved with the building of the machine to make sure you haven’t overlooked anything. Will your program mesh well with the mechanics, hydraulics, pneumatics, electrics and electronics? Will it make sense to the operators, and the maintenance people?

This is the best time to ensure that nothing has been overlooked - things like callibration and resumption from manual modes often affect large parts of the program, and if they’re not planned-for at this stage they’ll be difficult to add later.

If you’re programming for another company, make this one of the earlier deliverables ("Functional Specification") and sign off on it with the client. This makes it clear exactly what you’re planning to deliver, and what the client expects you to deliver.

[p 28] [p 1] [p 30]

$Date: 2004/12/28 05:32:12 $

[p 29] [p 1] [p 31]

HMI

Every control system has to fulfil the requirements of its working cycle but there are another two members of the "essential trio" - HMI and Safety [p 9] . They are always present, even when ignored by the author of the system.

Even the simplest control systems have to provide some form of Human Machine Interface. A few LEDs connected in parallel to output loads (or the sound of clicking relays) are a crude form of HMI. For anything a bit more complex - or anything that is to be used professionally - the HMI needs to be designed carefully. For many designs it may be biggest part of the project. It is also the part which sells, because it is visible.

It often helps if the HMI layout and function is designed in parallel with the control algorithm. The HMI has to be able to interface to the operation itself, but the design of HMI often reveals some aspects of operation which need to be controlled and which may not be otherwise obvious at early stages of the project.

A good HMI:

- is easy to understand,
- contributes to the safe and efficient running of the machine,
- is self documenting,
- includes on-line help (provides background information to operator),
- provides at least some data logging (you will be the first to need it - for debugging),
- may allow retrieving of logged data on screen or even replaying of logged cycle, and
- provides troubleshooting functions, to facilitate tracing of problems - the plant technician should never need to go to lower levels of software simply to find out which sensor burned out.

A good HMI should provide on-screen all information needed for control of operation, but low priority data may not be always visible. It is important to prioritize data for complex systems. Failure scenarios in particular have to be assessed, and the HMI should provide clear descriptions of problems and preferably offer possible solutions to the operator.

[p 29] [p 1] [p 31]

$Date: 2004/12/28 05:32:12 $

[p 30] [p 1] [p 32]

Testing

> I’m an idiot.. At least this one [bug] took about 5 minutes to find.. *(Linus Torvalds in response to a bug report.)*

Test rigs

When writing modular programs, one of the important things is to be able to test each section separately.

This means that often you will have to construct test rigs for each section that let you run it separately, making sure that it does what it is expected to do. This can be a lot of work, but pays off in tracing down problems - it’s much easier to find a problem the earlier it gets discovered.

Integration testing

The other part of testing is ensuring that the sections work well together.

Again, it is not a good idea to simply throw all the sections together and hope they work. As far as possible, always add one thing at a time. This cuts down on tracing the source of problems, because it will usually be connected with the section just added.

If the program is big enough to have sub-sections and sub-subsections, always integrate each section separately before putting them together. Again, the idea is to simplify tracking down problems.

Risk analysis

Under what circumstances can the software do damage to the machine or the product it is manufacturing? How much will this damage cost to rectify? These should form part of the test plan, including such techniques as simulating such situations (or inducing them under controlled conditions), to ensure that the software does indeed handle them satisfactorily.

If there is any way in which the software could (through action or inaction) cause a hazardous situation or harm a human, safety-critical software specialists need to be brought in, or such situations removed through hardware interlocks and the like.

[p 30] [p 1] [p 32]

$Date: 2004/12/28 05:32:12 $

[p 31] [p 1] [p 33]

Hints and Heuristics

A few snippets that may come in handy when writing programs...

- The human mind can keep track of 5-9 things at once (known as the "7+/-2 rule").

  Avoid having more than that many items anywhere in your design. Perhaps you can rearrange them into several smaller groups? If not, at least make sure that that part is as simple as possible in all other ways.

  This applies doubly so to the HMI, which will be used for a much greater amount of time and by a larger group of people. Thus, menus should generally have about seven items, procedures about seven steps, phone numbers about seven digits, and so on.

- 100% efficiency leads to infinite queues.

  Be careful with anything 100% (or zero). They can both lead to problems.

[p 31] [p 1] [p 33]

$Date: 2004/12/28 05:32:12 $

[p 32] [p 1] [p 35]

Modules

Modules are individual pieces of the MatPLC. Much like a real PLC, a particular MatPLC installation consists of several modules, plugged into a core (virtual backplane), working together to provide a useful application. There are I/O modules, logic modules, user interface modules etc. Most of this manual concerns the detailed instructions for one or another particular kind of module.

What modules are run is usually configured in the PLC section of the config [p 40] . It's also possible to start modules as separate programs, even when they aren't listed in the config; the plctest [p 55] tool is usually started this way.

Due to the high modularity of the MatPLC, most systems will consist of several modules - perhaps between three and a dozen. Even the simplest of demos typically have three or four modules.

Module groups

Modules are organized according to their function. This organization is purely to simplify orientation; once modules are running, there is no particular distinction between them (except that they do different jobs, of course).

Logic engines
: do the actual logic, decision making and calculation of the MatPLC.

I/O
: connect to the real world, or sometimes to a slave PLC over a bus or other connection.

HMI (or MMI)
: interact with the operator.

Module types

Modules come in three basic types.

generic
: These modules come with the MatPLC and are used without modification. An example is the DSP module - you specify in the `matplc.conf` file that you wish to use it and specify the inputs, outputs and tuning parameters for the PID loop (for instance).

  Most I/O modules also fall into this category.

specific
: These modules do not directly come with the MatPLC. Instead, they are created for a particular project using the tools provided. An example is the IL language [p 72] . You write a PLC program in

a file with the extension `.il`, which is converted into a module before starting.

The up-coming IEC 61131 translator will work the same way.

custom-made
: This is the ultimate in specific modules; if none of the existing modules is suitable for the job at hand, a completely new module can be written in the C language. Such a module can do anything, but it is much more difficult to write. It is essentially an extension of MAT itself.

[p 32] [p 1] [p 35]

$Date: 2004/12/28 05:32:10 $

[p 33] [p 1] [p 37]

Points

The points are the inputs, outputs, internal coils and registers of the MatPLC. Each point has a name by which it is known throughout the MatPLC.

Points may be 1-bit, corresponding to discrete inputs, outputs, internal/memory coils, flags etc, or multi-bit (up to 32-bit), corresponding to analog inputs, outputs, integer or floating point registers etc. For each point, there is exactly one module which is allowed to change it, to ensure that processing takes place in an orderly fashion.

Points are defined in the PLC section of the config [p 40] .

Example

For instance, in the basic demo [p 20] , there are the following points:

`L1`, `L2`, `L3`, `L4`
: the four "lights", which are changed by the chaser program;

`left`, `right`
: the left and right "buttons", which are changed by the kbd module and used by the chaser program to control the direction of movement; and

`quit`
: the quit "button", which is also changed by the kbd module; the plcshutdown module reads it so that it knows when to shut everything down.

The declaration in the config file looks like this:

```
point L1        "light 1" Chaser
point L2        "light 2" Chaser
point L3        "light 3" Chaser
point L4        "light 4" Chaser
point left      "<- (key L)" Kbd
point right     "-> (key R)" Kbd
point quit      "quit (key Q)" Kbd
```

All of the points are also displayed on the screen, by the vitrine module. No special access configuration needs to be done to allow this - any module may read any point it wishes; it’s only changing them that’s restricted.

In the modbus demo [p 21] , the `L1`, `L2`, `L3` and `L4` points are also sent to the Modbus slave.

[p 33] [p 1] [p 37]

$Date: 2004/12/28 05:32:10 $

[p 35] [p 1] [p 40]

Config

The config file is traditionally called `matplc.conf` (though you can give it any name you like). It determines what modules are running and any tuning parameters the modules require as well as configuring the MatPLC core, its global memory map and the like.

As such, the `matplc.conf` file is effectively the ‘main’ file in any installation: all options and settings are either in this file, or in some file it specifies.

The Configuration Editor [p 51] will be the easiest way to create and edit `matplc.conf`, once it is finished. For now, you will still have to do at least some editing with a text editor.

Example

There’s an example config file in the Appendix [p 153] . It may be useful to refer to it while reading this section.

Syntax

The config file is read line by line: for the most part, each line is interpreted separately, without regard to what’s on the previous or following lines, with the exception of sections.

Lines that begin with # are comments, and are ignored. Comments may also be placed on the end of a line. To put a # into a value, enclose the whole value with quote marks, `"`

Quote marks may generally be used to enclose things that otherwise wouldn’t be allowed in a value, like spaces or similar. Quotes may be included as `\"` and backslashes must then be `\\`

First line

We plan that in the future it’ll be possible to make the config files executable scripts. This will be indicated on the first line with the usual `#!` magic. As far as the MatPLC itself is concerned, that’s just a comment.

Sections

The config is divided into sections. There are two kinds of section names: one is simply prescribed, at present there’s only one of those, named `PLC`. In addition, when a module is started, it is always given a name, and this name is used as the config section to retrieve all settings. In this way, two modules of the same kind (for instance two Modbus modules, to run on different serial ports) can be started at once without interfering with one another by giving them different names.

Sections are indicated in two ways:

- using a `[section]` header on its own line, which gives the section for the following lines up to the next `[section]` or the end of the file, or
- by putting a `section:` prefix on the beginning of a line, which overrides the section only for this line. The next line is back to the section indicated by the last `[section]`, unless it has its own prefix.

Sections may be freely switched throughout the config files; the MatPLC sorts them out as it reads it from disk.

Single values

Single values are set in the config using the syntax:

```
name = value
```

In each section, each setting can only be given once, because it doesn't make sense to have two different settings for the same option. (Technically, it can be repeated, but only if they all have exactly the same value.)

Tables

Multiple values, known as "tables", are set using the syntax:

```
name value value...
name value value...
...
```

These may be interspersed with other settings and tables, even in separate pieces of a section or different files. When the config is read in, all the pieces of each table are joined together exactly as though it was given in one place.

Note that the first value in each row must start with an alphanumeric; if you need to start it with a punctuation symbol, enclose it in quotes. This is because the MatPLC needs to be able to distinguish tables from single values and so on.

File inclusion

Other files may be included in the config using the syntax

```
*include filename
```

This allows one to separate the config into various logical pieces, for instance by module, or to have various configurations of options with common pieces being in common files.

The included files may themselves `*include` others, and so on. Even circular `*include`s are supported and handled sensibly: each file will only be read once. Shell expansion is also done, so `filename` may include wildcards, environment variable substitution, and so on (however, backticks are

not allowed).

Sections do not continue from one file to another: each file must take care of its own sections. In effect, this basically means that each file must start with a `[section]` header, possibly after some comments.

[p 35] [p 1] [p 40]

$Date: 2004/12/28 05:32:10 $

[p 37] [p 1] [p 45]

Config - PLC section

The `[PLC]` section of the config is used to configure parameters related to the core which have to be the same in all the modules. Some other core-related parameters are module specific, and must be specified under those modules' sections - see the Common configuration [p 45] chapter.

Module and point tables

Almost every application will have to specify at the `module` and `point` tables. They are central to the functioning of the MatPLC.

`module`
: The `module` table defines which modules should be started by the matplc [p 50] program when it's told to start up the MatPLC. See the modules chapter [p 33] for explanation of modules.

  syntax:
  ```
  module <name> <file> [<options> ...]
  ```

  where:
  `name`
  : the name of the module. This name will be used in sections, module synchronisation, etc...

  `file`
  : the program that will be executed.

  `options`
  : other command line options that need to pe passed to the program.

`point`
: The `point` table defines the named points in the plc. See the points chapter [p 35] for explanation of points.

  syntax:
  ```
  point <name> <full name> <owner> [at <offset>[.<bit>]]
  [[i|u|f]<length>] [init <init_val>]
  ```

  where:
  `name`
  : Name used to refer to the point throughout the matPLC

  `full name`
  : More extensive description of the point. Currently, this is not used for anything; however, a diagnostic tool could use it to describe points to the operator.

  `owner`
  : Name of the module with write permission on the point - that is, the module that is allowed to change it.

`at`

the word ’at’. This indicates that the optional `<offset>` or `<offset>.<bit>` specification follows. Usually, this will be omitted and the MatPLC will allocate a slot for the point automatically.

`offset`

the word in the globalmap to be used

`bit`

the bit, in the word, that holds the point. If `offset` is specified, but not `bit`, `bit` defaults to 0. Note that a point must not overflow onto the next offset position (i.e. `bit` + `length`) <= 32).

`i | u | f`

one of the letters ’i’, ’u’ or ’f’. These are used to specify the type of the data in the plc point:

`i`	Signed integer, that is, whole numbers both negative and positive.
`u`	Unsigned integer, that is, whole numbers from zero up (no negative numbers).
`f`	Floating point; this really only makes sense with 32-bit points, otherwise known as `f32`. Floating point numbers use the standard IEEE representation.

The type specification is optional. There is no default; if the type is not specified, it is treated as unknown.

At the moment this is only used by the MatPLC code to figure out how to interpret the initial value of the point when parsing the `init` parameter. If the type is unknown, autodetection is attempted (as described below).

`length`

the size in bits of the point. The length is optional, defaulting to:

- 1 bit normally,
- 32 bits if `at <offset>` is used, and
- 1 bit if `at <offset>.<bit>` is used.

`init`

the word ’init’, indicating that the optional initial value follows. If it is omitted, the initial value will be 0 (off).

`init_val`

the initial value of the plc point. Interpreted according to the `i | u | f` specification; if the `i | u | f` is omitted, the initial value will be interpreted as follows:

- if it contains a decimal point (e.g. `1.1`) or an exponent (e.g. `1e9`), it will be taken as a a 32 bit float (`f32`);
- otherwise, if it has a leading ’+’ or ’-’ character, it will be interpreted as a `length`-bit signed integer (`i`);
- ohterwise, the value will be interpreted as a `length`-bit unsigned integer (`u`).

Only 32 bit floats, integers or unsigned integers are currently supported; however, shorter integers convert correctly.

Point alias table

The `point_alias` table defines aliases to the named points. It is optional - in many applications, there will be no need for aliases.

As far as the rest of the MatPLC is concerned, a point alias is exactly the same as a point. Aliases can be used for two purposes:

1. to have one point referred to by two names, and
2. to have one point refer to particular bit(s) of another.

syntax:
`point_alias <name> <full name> <orig_pt> [<bit> [<length>]]`

where:

`name`
: Name used to refer to the point alias throughout the MatPLC.

`full name`
: More extensive description of the point alias - as for a normal point.

`orig_pt`
: The name of the original point.

`bit`
: The first bit of the original point that this alias will reference. This is optional - by default, the alias refers to the whole original point.

`length`
: the size in bits of the alias point. The alias must not overflow outside the original point, so that `bit` + `length` <= (length of `orig_pt` point). The `length` is optional; if `bit` is specified but `length` is not, the alias will refer to that single bit (that is, `length`=1). If neither is specified, the alias refers to the whole original point.

The owner and initial value are specified on the original point; they don't need to be set here.

Single values

All of these settings have reasonable defaults which will work for the vast majority of applications.

`magic_bit_aliases`
: This is a feature to simplify access to individual bits of multi-bit points. By default it is off, to reduce confusion, but it can be activated by setting it to 1. When active, any time you have a point called `foo`, it will create aliases `foo.0`, `foo.1` and so on up to the length of the point.

  This setting can also appear in the section of a particular module, creating the bit aliases only for that module.

`max_modules`
: The maximum number of simultaneously active modules the MatPLC will ever have.

If this parameter is not specified, a default value (currently 20) is used. Valid values are non-negative integers. The allowable maximum depends on the operating system currently being used and the number of places in the synchronisation Petri Net (see the synch chapter [p 46]).

`confmap_key`

The key to use for the confmap shared memory. This is actually the same parameter configured through --PLCplc_id=xxx. The command line argument takes precedence over the value configured in this file.

If this parameter is not specified either way, the default value (currenty = 23) is used. Valid values are positive integers or 0. If 0 is specified, then a random key is chosen.

`confsem_key`

The id of the sempahore set used by the CMM.

If no value is specified, then the default is used (currently 0). Valid values are positive integers or 0. If 0 is specified, then a random key is chosen.

`confmap_pg`

The size of the confmap given in number of memory pages.

If this value is left unspecified, the default value (currently 2) is used. The size of each memory page is given by PAGE_SIZE defined in asm/page.h or sys/user.h (which is the correct include?) (for linux on intel (?) PAGE_SIZE is 4 kBytes).

`globalmap_key`

The key to use for the globalmap shared memory segment.

If no value is specified, then the default is used (currently 0). Valid values are positive integers or 0. If 0 is specified, then a random key is chosen.

`globalsem_key`

The key to use for the globalmap semaphore set used by the GMM.

If no value is specified, then the default is used (currently 0). Valid values are positive integers or 0. If 0 is specified, then a random key is chosen.

`globalmap_pg`

The size of the globalmap given in number of memory pages.

If this value is left unspecified, the default value (currently 2) is used.

`synchsem_key`

The id of the sempahore set used by the SYNCH sub-system.

If no value is specified, then the default is used (currently 0). Valid values are positive integers or 0. If 0 is specified, then a random key is chosen.

Synch and Real-time

Some of the synch config and real-time config are also in the PLC section; they are covered in the Synch chapter [p 46] and RT chapter [p 47] respectively.

Examples

Example 1: Absolute Minimum Configuration

```
[PLC]
# we need some points:
point P1  "full name 1" module1       # 1-bit point
point P2  "full name 2" module1   5   # 5-bit point
point P3  "full name 3" module2  32   # 32-bit point

[PLC]
# We also need some module to execute:
module module1 /matplc/logic/...
module module2 /matplc/logic/...

[module1]
# Here go the module-specific configurations for module1

[module2]
# Here go the module-specific configurations for module2
```

[p 37] [p 1] [p 45]

$Date: 2005/06/19 04:52:05 $

[p 40] [p 1] [p 46]

Common configuration

This section of the manual is not yet written. It will explain the settings common to all modules.

[p 40] [p 1] [p 46]

$Date: 2004/12/28 05:32:10 $

[p 45] [p 1] [p 47]

Configuration of synch

This section of the manual is not yet written. It will explain how to configure the synch. Some of these are settings in the `[PLC]` section, others are common to all modules, but they're all collected here because they logically belong together.

Introduction

There is no "global cycle" in MatPLC unless the user specifically configures one. By default, all modules run asynchronously, with semaphores making sure they don't step on each others' toes.

Explanation from a PLC background: the MatPLC reduces scan time at the expense of latency - at the top/bottom of the scan, it simply uses the latest available values rather than waiting for fresh ones. This is by default.

Using synch, the user can configure a global scan which interleaves scans of the ladder with refreshing the I/O, classical PLC style.

For modules like a modbus slave, one would normally leave it asynchronous, serving requests as they come in.

Similarly for any other modules that don't really belong in the global cycle. Perhaps they might even have a mini-global-cycle of their own - for instance, an I/O module and a corresponding DSP module might be synched together but not to anything else.

...

Not yet written - two methods, simple method, advanced method.

[p 45] [p 1] [p 47]

$Date: 2004/12/28 05:32:10 $

[p 46] [p 1] [p 50]

Configuration of real-time features

Some parts of this page describe proposed functionality. Not all of the functionality described in this section of the manual is available yet!

Introduction

All applications have requirements on timing, but for most kinds of computer programs they are not particularly strict. If a word-processor unexpectedly takes several seconds to do some operation, it is not a problem. For PLC programs, however, they are often very important. The mechanisms used to deal with the timing requirements are collectively called "real-time".

MatPLC modules will be able to run in one of three modes: normal, soft real-time and hard real-time. Different modules can run in different modes at the same time.

normal
: No action taken to guarantee response time. This is suitable for demos and low-priority monitoring, but not much else. This is the default. Status: ***available now***.

soft real-time
: The module has absolute priority over other programs, but not over the operating system itself, so there are no guarantees. This is the best that can be done with a generic linux kernel. Status: ***available now***.

hard real-time
: Provides timing guarantees, but requires a special kernel modules, and some care in writing the program. (Note: this does not necessarily mean that it’s fast; merely that the timing is accurate when it needs to be.) Status: ***not yet implemented***.

The MatPLC can be compiled in two ways: N/S, where Normal and Soft-RT is available, and N/S/H where all three are available but the special kernel is required. The default is N/S.

Real-time and synch

There is some interaction between the real-time stuff and synch; fortunately, the rule is simple: never make a synch arrow from a low priority module to a high priority one.

You can synch between equal-priority modules, and you can make synch arrows from high priority modules to lower priority ones; but never the other way around.

Modules

In general, there's no change to a module regardless of whether it's running normal, soft real-time or hard real-time

However, there are some modules which may not run in hard real-time mode, because of the nature of their work. For instance, if a module needs to write to a file, it has to be normal or soft real-time. For the generic and specific modules, this is noted in the manual where appropriate. For custom modules, the module-writer needs to be careful: if any system calls are used, the module may not be run in hard real-time mode.

Swapping

As you can imagine, being swapped out to disk will give a module a sizeable timing glitch. The first step, then, is to make sure it doesn't happen. The downside is that only programs that run as root can do this. (**FIXME** is suid-root enough? capabilities?)

The setting `rt_memlock` in the `[PLC]` section controls this. When it's 1, all modules will be locked into memory. (Due to the way the MatPLC works, it's not really sensible do this on a per-module basis.)

When compiled N/S/H, this option is always on (and only root can start modules).

Status: this setting is **implemented** but not tested

Soft-RT

The setting `rt_softpri` controls this. This is a per-module setting, with a default in the `[PLC]` section. When it's greater than 0, the module is made soft-rt with the given priority. The allowable range is 1-99, with 99 being the highest.

Only modules started by root will be able to go soft-rt.

Status: this setting is **implemented** but not tested

Low-latency kernels

The timing performance of Normal and Soft-RT can be improved by using a low-latency kernel. Such a kernel switches between processes more often than the generic one, thus reducing the latency in any one process. The cost is that all this switching does reduce overall performance slightly, which is why this tuning parameter is set differently for general purpose kernels.

Similarly, different versions of the kernel have different default schedulers, which have different features. For instance, see this article on LinuxDevices.com about the linux 2.6 improvements.

Hard-RT

When MatPLC is compiled N/S/H, it uses RTAI LXRT to obtain hard-realtime. Modules can still run in normal and soft-RT, but they will all be LXRT tasks, which means that they must be started by root and memory locking is always on.

The settings are **FIXME**

The caveats are **FIXME** (syscalls drop the module to Soft-RT, debugging not available, etc)

Status: this functionality is **not yet implemented**

Better than LXRT MatPLC

MatPLC, like any general purpose product or framework, is a compromise between various aspects. If timing accuracy is paramount, one will have to skip the niceties and write a (hopefully small) part of the application as a kernel-space hard-RT task doing direct I/O. Writing and debugging such code is beyond the scope of this manual.

Much tighter results still can be obtained by pushing the speed-sensitive parts to FPGA or other hardware; John Storrs's LME Craftsman project (**FIXME** - add URL) takes this approach, giving nice, clean, jitter-free oscilloscope traces.

As far as the MatPLC is concerned, these would simply be "smart I/O". For instance, the MatPLC could command direction, speed and number of steps, and the kernel-task or FPGA would generate the stepper motor train.

[p 46] [p 1] [p 50]

$Date: 2005/05/15 14:21:53 $

[p 47] [p 1] [p 51]

matplc

This is MAT PLC’s main "control" program, used for starting the MatPLC and shutting it down.

This page of the manual is not really written yet, there’s just the usage summary here. **FIXME**

```
usage: .../matplc  {-{c|g} [file_name] | -{r|h} [] | -{s|l|d} [PLC options]
          c: Check matplc config file syntax. fname defaults to matplc.conf
          g: Go. Start the MatPLC. fname defaults to matplc.conf
          s: Shutdown the MatPLC.
          r: Place the MatPLC, or the <module> if specified, in RUN mode.
          h: Place the MatPLC, or the <module> if specified, in STOP mode.
          d: Dump the MatPLC configuration map.
```

Source code note

If you’re looking for the source code to the `matplc` program, it’s actually in the `lib/util` directory, even though the program itself is in `tools/run`. This is simply for historical reasons.

[p 47] [p 1] [p 51]

$Date: 2004/12/28 05:32:12 $

[p 50] [p 1] [p 53]

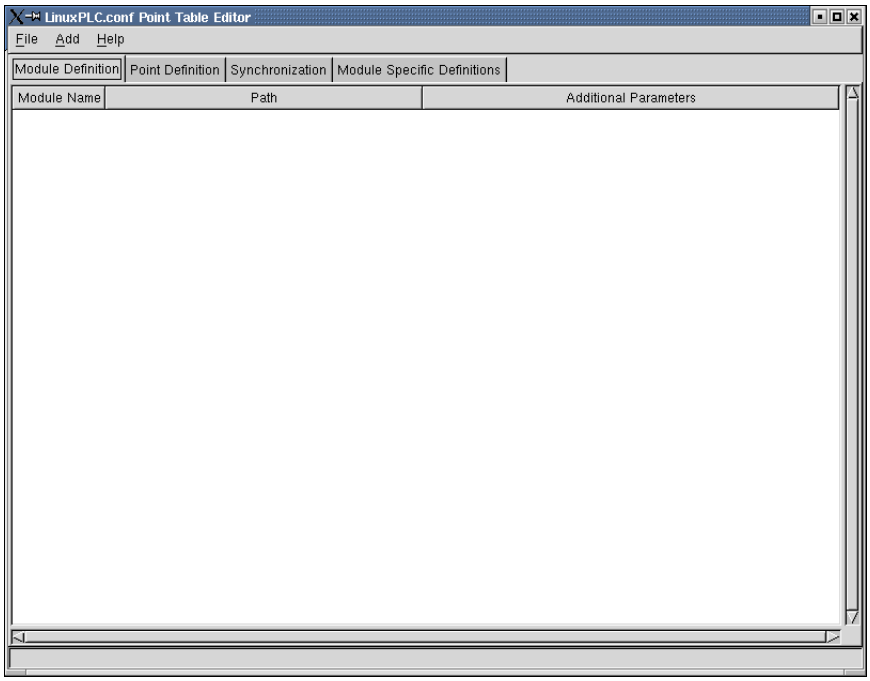
CONFIGURATION EDITOR

This module is Copyright © 2002 Joe Jansen and is distributed under the GPL license [p 165] .

Introduction

The configuration editor is intended to be an easy to use method to create and edit the configuration file for the MAT/Linux PLC. This is typically named matplc.conf, although other files can be included.

config-edit consists of a single window with the main window pane consisting of several notebook tabs that allow various parts of the file to be viewed and edited.



The notebook tabs are labeled Modules, Points, Synchronization, and Module Specific Definitions. At this time, synchronization and Module Specific Definitions have not been implemented.

Future versions will implement a seperate notebook tab for each module to define specific keyword/value pairs.

For details on the inner workings of the matplc.conf file, refer to the documentation for that file. This configuration editor will build the module table, point table, and synchronization tables in the PLC section, as well as create and specific sections needed for the individual modules.

[p 50] [p 1] [p 53]

$Date: 2004/12/28 05:32:12 $

[p 51] [p 1] [p 54]

CONFIGURATION EDITOR

This module is Copyright © 2002 Joe Jansen and is distributed under the GPL license [p 165] .

Module Definition

The first notebook tab is used for defining the different PLC modules, or components that will be used in this implementation. This is not only for modules such as the logic engines, but includes HMI, I/O drivers, and all other components that will make up the final PLC.
Modules are defined by selecting ADD -> Modules from the menu.

FIXME: Figure `figure2.png` is missing...

Module definition has three components. Name, path, and additional parameters. The first two are mandatory, the third is optional.

Name: This is the name for the module that will be used to reference it throughout the rest of the configuration file, and in other modules as necessary. It does not need to be the name of the executable.

Path: This is the path to the executable file, including the name of the executable. There is a browse button that can be used to select the executable file.

Additional Parameters: This is comand line parameters that need to be passed to the executable at startup. This is different from module specific keyword/value pairs that are defined in a different section.

[p 51] [p 1] [p 54]

$Date: 2005/10/11 12:34:15 $

[p 53] [p 1] [p 55]

CONFIGURATION EDITOR

This module is Copyright © 2002 Joe Jansen and is distributed under the GPL license [p 165] .

Point Definition

The point table is a central part of the Linux PLC. This is where all of the tags are defined for communicating betweeen modules, including external I/O. Every point has exactly one ’owner’ that has the ability to set the value of the point. For example, external inputs are typically owned by the module that controls the I/O device.

Points are added by selecting Add -> Points from the menu.

FIXME: Figure `figure3.png` is missing...

Mandatory fields are Name, Owner, and Point Type.
ID (Name): This is the identifier that will be used to access this data point. This must be a unique name for all data points. Spaces are not allowed
Description: This is an informative field for the user. This can be a multi word descriptor. Modules such as logic editors can use this for documentation. Spaces are allowed.
Owner: The owner field is implemented as a drop down menu item. The menu is populated using the names of the modules that have been entered in the module list. The module that is selected here is the only module that has write access to this point.

[p 53] [p 1] [p 55]

$Date: 2005/10/11 12:34:15 $

[p 54] [p 1] [p 57]

mattest / plctest

This is MatPLC's Swiss Army Knife. It can be used to get and set the values of points, wait for a particular point or synch point from a script, and a few other miscellaneous debugging things.

Note: this tool has been recently renamed: it used to be called "plctest" but is now called "mattest". The functionality is unchanged.

For using it from a script, it's probably a good idea to use the `-q` option, which switches it to *quiet* mode. In quiet mode, plctest only outputs the requested data (and any errors).

Getting and setting points

To read a point, use the -g option, like this:
`plctest -g pt_name`

If the point is a floating-point number, you'll get some strange number. To convert it to a float, use the -f option, like this:
`plctest -f -g pt_name`

It's also possible to set points, but note that only the owner of a point can set it. You can get plctest to pretend to be the owner, but if the real owner is already running, it will overwrite your value in short order. So this is only useful in limited circumstances. The syntax is similar to the above, though:
`plctest -s pt_name -v value`
`plctest -f -s pt_name -v value`

Display all points

As a debugging tool, it's often useful to be able to see what's happening inside a PLC. While a full-fledged displayer is not yet written, plctest contains a quick, simple one.

On screen

This simply displays all the points in four columns and repeats until you kill it (Ctrl-C).
`plctest -d`

HTML (Web / Intranet)

This feature is also available via a web browser. If you run
`plctest -dh`
it will output a HTML table with the point names in the left column and the values in the right column. So you might have a CGI script, perhaps a bit like this:

```
#!/bin/sh
cd .../mat/demo/basic
echo Content-type: text/html
echo ""
../../tools/run/plctest -dh
```

Or, more sophisticated:

```
#!/bin/sh
cd .../mat/demo/basic
echo Content-type: text/html
echo Refresh: 10
echo Pragma: no-cache
echo ""
cat preamble
../../tools/run/plctest -dh
cat postamble
```

XML

Similarly, the data can be output in the XML format. The command
`plctest -dx`
will output simple XML containing the points (names and values).

Waiting...

plctest can wait for a synch point or until a particular point is on. This can be useful in scripts, to avoid looping in the script.
`plctest -u` *`pt_name`*
`plctest -w` *`synchpt_name`*

Miscellaneous

plctest can also run an empty scan loop. This can be used to practice synch, as it prints out loop counts each time through (unless you use the `-q` option).
`plctest -l`

[p 54] [p 1] [p 57]

$Date: 2005/05/07 07:43:45 $

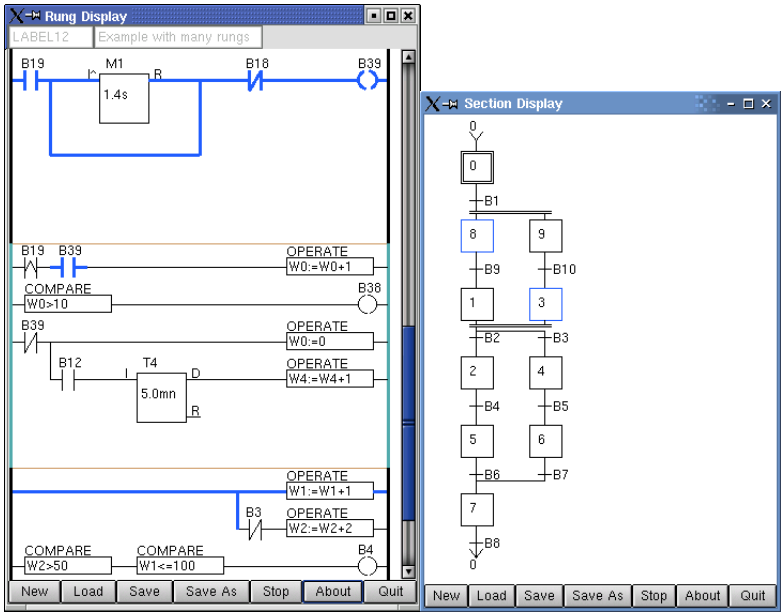
[p 55] [p 1] [p 58]

Classicladder

Classicladder is a graphical stepladder and GRAFCET editor and executive module. It is a separate project led by Marc Le Dourain, and it can be obtained from his website,
http://www.multimania.com/mavati/classicladder

The website also contains the relevant documentation.

Any enhancements, bug reports, bug fixes and so on (including those relating to the MatPLC integration) should be submitted to Marc.



[p 55] [p 1] [p 58]

$Date: 2004/12/28 05:32:11 $

[p 57] [p 1] [p 68]

Digital Signal Processing

Introduction

The DSP module carries out various sorts of floating point operations, including arithmetic, speed and acceleration limiting, PID control, scaling, alarming and so on. It can also handle integer registers by converting them to floating point first.

The operations to be carried out by the DSP module are specified directly in the matplc.conf file (or in a separate file, using `*include` *`filename`*), in the `fblock` table.

The `fblock` table has the usual stepladder semantics: at the top, register values and contacts are read from the rest of the MatPLC (inputs, HMI, or other logic modules). The table is then scanned from top to bottom, with the operations being done in order. At the bottom of the table, register values and coils are written to the MatPLC core, to be used by the rest of the system (outputs, HMI or other logic modules).

Configuration

The dsp config has one main parameter, and one table (`fblock`).

The `fblock` table is used to configure the function blocks the dsp module will execute.

The main parameter is:

```
  out_time_pt = <plc_pt>
where plc_pt is where the dsp will store the current time
out_time_pt = time
```

The scan_period parameter is valid for any module, but it will probably be used extensively by the dsp module.

```
scan_period = x
  time in seconds
scan_period = 0.1
```

The `fblock` table is used to configure the function blocks the dsp module will execute. The function blocks are executed in the same order in which they are configured (ie top to bottom). In the usual stepladder way, inputs are read once at the top of the table and results are not available to other modules until the bottom of the table.

Supported function block types

typeconv
: copy the value in an input point in one format, to an output point in another format. Supported formats are i32 (32 bit int), u32 (32 bit unsigned int), and f32 (32 bit float). All other function blocks use input and output points in the f32 format.

add
: Add (with a specified multiplier) a maximum of 10 PLC points.

mult
: Multiply a maximum of 10 PLC points.

pow
: Raise a plc point to the power of x (x being a parameter).

pid
: Implements an open loop PID function. For closed loop controllers, use an add block to close the loop.

filter
: Filter the input, using an iir (infinite impulse response) filter.

ramp
: Limit the maximum speed (dx/dt) and aceleration (d2x/dt2) with which the value of a given variable (x) may change.

nonlinear
: Implements a deadband and limiting function.

alarm
: Compares the value of a PLC point to up to a maximum of 10 limit values, and sets the value of output points accordingly.

multiplexor
: Copy one of the input points to the output point. The input point that is copied is dependent on the value of a control point.

Typeconv block

```
#fblock typeconv <in_pt1> <in_pt1_type> <out_pt1> <out_pt1_type>
               [<in_pt2> <in_pt2_type> <out_pt2> <out_pt2_type>] ...

<in_pt_type>
<out_pt_type> : How to interpret the bits in the previous point.
                = {i32 | u32 | f32}
```

Add block

```
#fblock add <out_pt>     <in_pt1> <in_pt1_mult> [<in_pt2> <in_pt2_mult>] ...

 out_pt = (in_pt1 * in_pt1_mult) + (in_pt2 * in_pt2_mult) + ...

 Note: currently a maximum of 10 in_pt are supported.
```

Mult block

```
#fblock mult <out_pt>     <in_pt1> <in_pt1_ofs> [<in_pt2> <in_pt2_ofs>] ...

 out_pt = (in_pt1 + in_pt1_ofs) * (in_pt2 + in_pt2_ofs) * ...

 Note: currently a maximum of 10 in_pt are supported.
```

Pow block

```
#fblock pow <out_pt> <in_pt> <in_pt_pow>

where:
  out_pt    : matplc point to be used as output
  in_pt     : matplc point to be used as input
  in_pt_pow : raise in_pt to in_pt_pow (f32 value)

                  in_pt_pow
  out_pt = (in_pt)
```

PID block

`fblock pid <in_pt> <out_pt> [<P> [<I> [<D>]]] [max_out <upper_limit>] [min_out <lower_limit>] [man_mode <manual_mode_pt>] [man_out <manual_value_pt>]`

This block implements a parallel PID controler.
`out_pt = P*in_pt + I*integral(in_pt, dt) + D*din_pt/dt`

The output is guaranteed to stay within the configured limits (lower_limit <= output <= upper_limit) in automatic mode. This is achieved by adjusting the integral, which means that the pid will not ’wind up’ when the output is saturated.

The man_mode and man_out settings work together to provide bumpless manual-to-automatic transfer. See the bumpless transfer section [p 109] for more details.

When man_mode_pt is ON, the block is in manual mode:
`out_pt = manual_value_pt`
In manual mode, the integral is adjusted in such a way that when man_mode_pt goes OFF, the PID block will smoothly take over from the operator.

Note that manual mode ignores the upper and lower limits; if necessary, pass the operator's setting through a nonlinear block to clip it to the limits (and perhaps also through the ramp block). This also means that if the operator sets the process outside the limits and switches to automatic, there will be a bump as the PID immediately jumps to within the limits.

Also note that bumpless transfers are only supported when I is non-zero.

setting	meaning	default value
P	proportional coefficient of PID	1
I	integral coefficient of PID	0
D	derivative coefficient of PID	0
upper_limit	maximum output value for automatic mode	none
lower_limit	minimum output value for automatic mode	none
manual_mode_pt	manual/automatic selection	none (always automatic)
manual_value_pt	value to be output in manual mode	none (always zero)

Filter block

```
fblock filter iir [<C> [<A1> [<A2> [<B1> [<B2>]]]] ...
```
or
```
fblock filter <filter_type> <filter_shape> <gain> <passband_tol>
<stopband_tol> <f1> <f2> [<f3> <f4>]
```

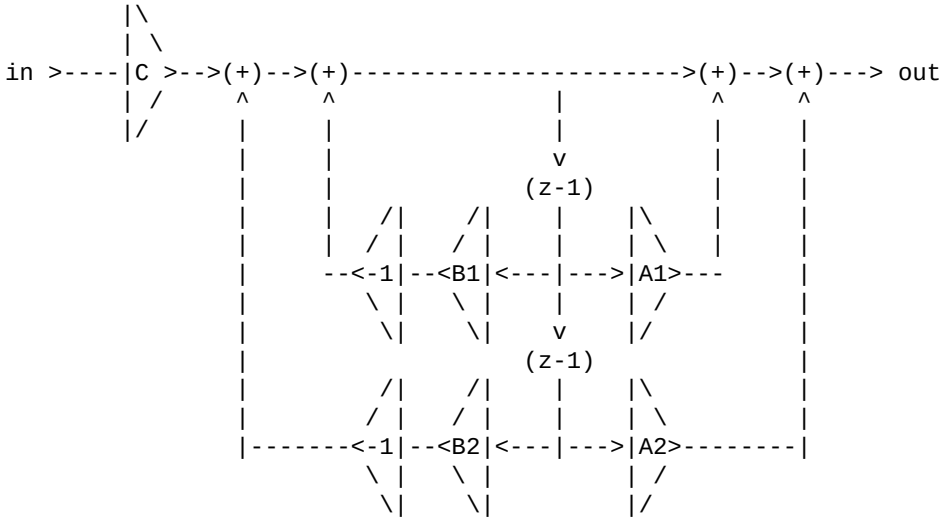
Where:

C, A1, A2, B1, B2	second order iir filter parameters directly	Please see below for an explanation of what an iir filter is. These five parameters completely specify the filter, so no further parameters are necessary.
filter_type	{butterworth | chebyshev | elliptic}	This method is used to configure the filter as an equivalent to an analog filter. At setup these parameters are transformed into the equivalent iir filter, which will be executed at run-time.
filter_shape	{lowpass | highpass | bandpass | bandstop}	NOTE: These transformations have not been completely debuged. lowpass and highpass *should* be working correctly. bandpass and bandstop are known to be buggy.

gain	the gain of the filter for the passband frequencies	For	lowpass,	gain = g(0)	i.e. gain at f = 0 Hz
			highpass,	gain = g(F/2)	i.e. gain at f = F/2 Hz
			bandpass,	gain = g(fp1/2 + fp2/2)	i.e.jgain at f = (fp1 + fp2) / 2 Hz
			bandstop,	gain = sqrt [g(0)*g(0) + g(F/2)*g(F/2)]	
		(F is sampling frequency)			
passband_tol (stopband_tol)	max(min) atenuation for pass(stop)band	maximum(minimum) atenuation, in positive dB, for the passband(stopband) frequencies			

fX	frequency limits in Hz	NOTE: The meaning of f1, f2, f3 and f4 depends on the filter shape. The idea is to have valid frequencies always in increasing value. For lowpass f1 : passband frequency limit f2 : stopband frequency limit highpass f1 : stopband frequency limit f2 : passband frequency limit bandpass f1 : lower stopband frequency limit f2 : lower passband frequency limit f3 : higher passband frequency limit f4 : higher stopband frequency limit bandstop f1 : lower passband frequency limit f2 : lower stopband frequency limit f3 : higher stopband frequency limit f4 : higher passband frequency limit

The filter block implements X second order iir filters in series. A second order iir filter:



where (z-1) is a delay block, i.e., in the z transform, z to the power of -1.

Ramp block

```
fblock ramp <in_pt> <out_pt> [dxdt <xx>] [pos_dxdt <xx>] [neg_dxdt
<xx>] [d2xdt2 <xx>] [pos_d2xdt2 <xx>] [neg_d2xdt2 <xx>]
```

Where:

in_pt	:	the plc_pt where the input value (x) is stored
out_pt	:	the plc_pt where to store the output value
dxdt	:	maximum speed with which x may change (both positive and negative changes).
pos_dxdt	:	maximum speed with which x may rise. Defaults to no limit.
neg_dxdt	:	maximum speed with which x may fall. Defaults to no limit.
d2xdt2	:	maximum speed with which dx/dt may change (both positive and negative changes).
pos_d2xdt2	:	maximum speed with which dx/dt may rise. Defaults to no limit.
neg_d2xdt2	:	maximum speed with which dx/dt may fall. Defaults to no limit.

Nonlinear Block

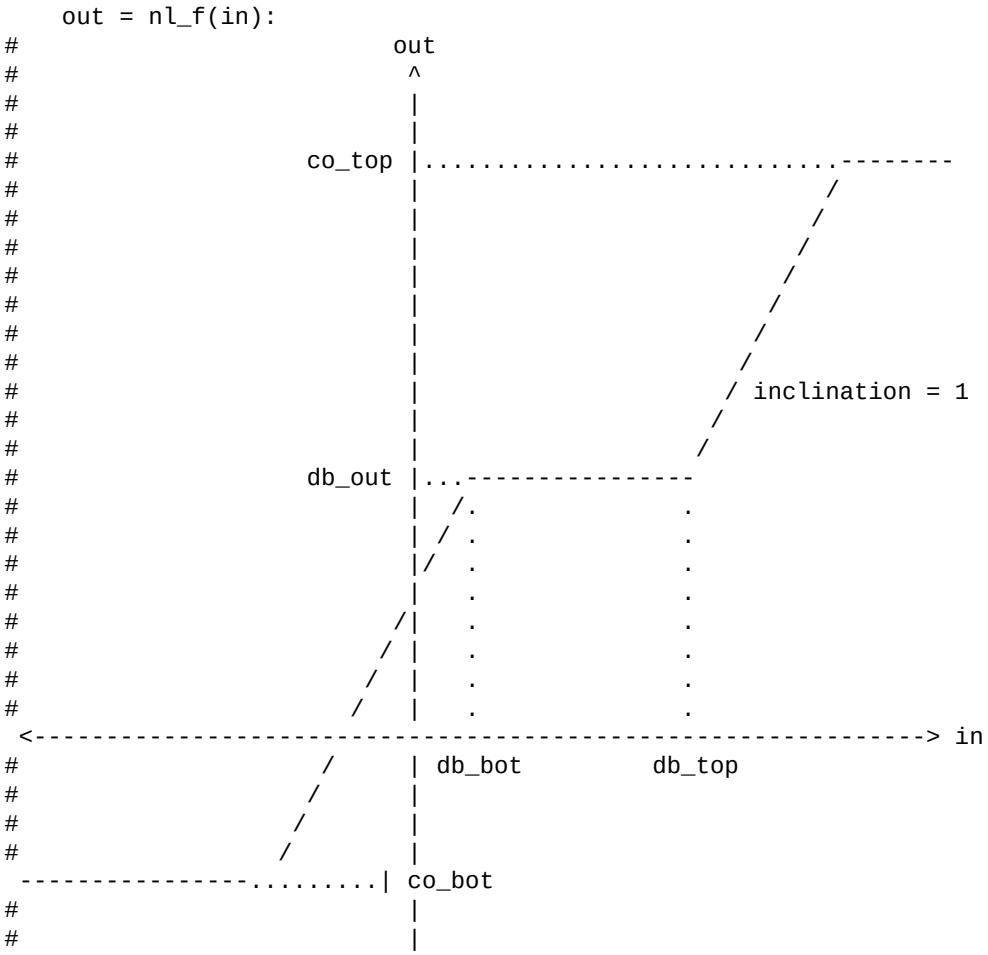
```
fblock nonlinear <in_pt> <out_pt> [cutoff_top <xx>] [cutoff_bot <xx>]
[deadband_top <xx>] [deadband_bot <xx>] [deadband_out <xx>] [gain <xx>]
```

Defaults are:

cutoff_top	:	f32_MAX
cutoff_bot	:	-f32_MAX
deadband_top	:	0
deadband_bot	:	0
deadband_out	:	0
gain	:	1

This function block implements a nonlinear block supporting both a deadband function centered around a configured offset, and a limiter. In addition, it also allows for a linear gain and offset to be applied to the output of it’s nonlinear function. These last two are essentially to reduce the number of function blocks eventually required to implement a specific global user function.

The nonlinear part, implements the following function:



```
#                           |
#                           v

#       (co = cutoff      db = deadband)
```

The linear part implements the following function:

```
out = l_f(in) = in * gain
```

The output of the complete nonlinear block is:

```
out =  nl_f( in * gain )
```

Alarm block

```
#fblock alarm <in_pt> {true_val|abs_val} <out_pt1> <comp1> <limit1> [<out_pt2> <comp2> <limit2>] ...
where:
  in_pt   : matplc point to be used as input for the alarm block
  true_val: use the value in in_pt without any changes
  abs_val : use the absolute value in in_pt for determining the alarms
  out_pt  : matplc point to be used as output for the alarm block
  limitl  : f32 value used for the alarm comparison
  comp    : specifies when the alarm should be set.
            one of:
              {less | lt | smaller | st |
               less_or_equal | le | smaller_or_equal | se |
               greater | gt | greater_or_equal | ge |
               equal | eq | not_equal | ne}
             Note: less, lt, smaller, and st, are all equivalent
                   greater and gt are equivalent
                   greater_or_equal and ge are equivalent
                   etc...

Example:
fblock alarm in_pt true_val out_1 10 lt out_2 10 gt out_3 20.55 eq
 (consider in_pt_val the value currently stored in the in_pt PLC point)
  then the above config line will have the efect of:
   - out_1 being true (1) when in_pt_val < 10, and (false) 0 otherwise
   - out_2 being true (1) when in_pt_val > 10, and (false) 0 otherwise
   - out_1 being true (1) when in_pt_val = 20.55, and (false) 0 otherwise
```

Multiplexor block

```
#fblock multiplexor <out_pt> <ctrl_pt> <in_pt1> [<limit1> <in_pt2>] ...

where:
  out_pt : matplc point to be used as output for the multiplexor block
  ctrl_pt: matplc point used to decide which input to copy to the output
  in_ptX : matplc point to be used as input for the multiplexor block
  limitX : limit value at which the output switches from one input to another

  out_pt = in_pt1 -> if (ctrl_pt <  limit1)
  out_pt = in_pt2 -> if (ctrl_pt >= limit1) AND (ctrl_pt < limit2)
  out_pt = in_pt3 -> if (ctrl_pt >= limit2) AND (ctrl_pt < limit3)
  ...
```

[p 57] [p 1] [p 68]

$Date: 2004/12/28 05:32:11 $

[p 58] [p 1] [p 72]

IEC ST/IL compiler

Introduction

One of the tools provided with the MatPLC is a compiler for the IEC 61131-3 IL and ST textual languages. Basically, this tool transforms a program written in ST and/or IL into C++ code that, when further compiled and linked to the MatPLC libraries, becomes a MatPLC module.

To write an ST or IL program, use a text editor and save the file with whatever name you wish. We sugest you use <prog_name>.st or <prog_name>.il as this is the convention that has been followed in the MatPLC demo directories. The IEC 61131-3 standard simply uses files names <prog_name>.txt in their examples, so you may want to use that instead.

Turning the ST/IL file into a module

The compiler accepts as its only parameter the name of the file to be compiled, using stdin if no filename is given. The resulting C++ code is sent to stdout.
e.g.:

```
$iec2cc oven.st > oven.cc
```

or

```
$cat oven.st | iec2cc > oven.cc
```

The resulting C++ code will need to be further compiled by a C++ compiler. Note that it needs to include the standard `$MATPLC/lib/plc.h` header file, as well as `$MATPLC/logic/iec/stage4/generate_cc/plciec.h`

Once compiled, the resulting object code will have to be linked with the standard matplc library `$MATPLC/lib/matplc.[so|a]`. We sugest that you use the working demo programs and makefiles in the `$MATPLC/demo/basic_iec` directory as a starting point.

The resulting executable code may be run as any other MatPLC module, including all the standard --PLC<plc_option> command line switches.

The ST and IL languages

Currently this manual does not include any description of the ST and IL languages. Users not familiar with the IL and ST programming languages are invited to read a book on these languages.

A useful on-line reference is IEC 61131-3: Programming Industrial Automation Systems.

Note, however, that most books currently available on programming with the IL and ST languages refer to the first version of these languages, whereas the iec2cc compiler was built based on the 2nd verion of the standard. More precisely, the publicly available draft version of the standard, dated 10th december 2001, was used.

Current State of the IEC Compiler

The iec2cc compiler as it stands is still very incomplete. It currently parses the complete syntax of the IL and ST languages, so any syntax errors in the program being compiled are caught early on. Nevertheless, error messages have not yet been written, so if the iec2cc compiler stops or simply runs into an infinite loop, this is because you have a syntax error somewhere in your IL/ST code.

Note however that not all syntax is yet suported when generating the equivalent C++ code. Aditionally, semantic checking is not yet performed, so most semantic errors (e.g. storing a TRUE value in a variable of type INT) in your programs will not be caught by the iec2cc compiler. Many semantic errors will nevertheless result in semantic errors in the resulting C++ code, so they will probably be caught later on by gcc, but this is not guaranteed.

Writing an IL/ST Program

Writing a MatPLC module in IL and/or ST for the iec2cc compiler is a question of merely writing all the program organization units (i.e functions, function blocks, programs and configurations) you require.

However, only the first configuration will be used by the iec2cc compiler. Remaining configurations will simply be ignored. Currently, this configuration may only contain a single program attributed to a single task.

For e.g.

```
CONFIGURATION config1  VAR_GLOBAL    varint : INT := 99;    varreal : REAL := 99.9;  END_VAR  TASK t1(INTERVAL := t#20ms, PRIORITY := 2);  PROGRAM foo WITH t1: PROG1(param1 := varint, param2 := varrea
END_CONFIGURATION
```

This implies that at the moment you will be writing a single program type, since a single instance of a single program type is currently supported.

The remaining functions and function blocks may be written in either ST or IL, as long as each function or function block uses a single language. The iec2cc compiler automatically distinguishes which language is being used.

Before calling a function, and before a function block or program instance is defined, the referenced function/function block/program must be defined. This means that when you can call a function, or create a function block instance, that function or function block instance must appear earlier in the file being compiled.

Note that your program may be divided into several files. A file may include another anywhere in the code using the syntax: `(*#include "<filename>"*)`

Note the lack of spaces between ’(*’ and ’#include’, as well as between the last ’"’ and ’*)’. This syntax will be interpreted as a comment by other IL/ST compilers, but will probably nevertheless be changed to conform to the ’pragma’ directive as defined in the IEC 61131-3 standard.

Accessing MatPLC Points

Accessing the MatPLC points from within IEC IL and ST programs is achieved using located variables. However, unlike the IEC standard that restricts the location of variables to names such as ’%I.0.2’, ’%Q.9.34’, etc., we allow the location to be any identifier that may also be used as a IL/ST variable name. For example,

```
VAR_GLOBAL  leds AT %Lights : INT := 1;  left_bt  AT %left_bt  : BOOL;  right_bt AT %right_bt : BOOL;END_VAR
```

From this declaration onwards, accessing the ’leds’ variable from within the IL/ST program will directly access the ’lights’ MatPLC point.

If the MatPLC point is wider (has more bits) than the data type of the corresponding ST/IL variable, writing to this variable will set the unused bits to zero, while reading the variable will ignore the extra bits.

Likewise, if the MatPLC point has less bits than the data type of the corresponding ST/IL variable, then writing to the variable will essentially discard the higher valued bits, while reading from the variable will set the higher valued bits to zero.

Note however that MatPLC points may have names such as ’temp__alarm’, that are not valid ST/IL identifiers. In this case it is not yet possible to map them to ST/IL variables (this issue will be resolved at a later date), so we sugest that an alternative name is given to these MatPLC points, or an alias be established (see the ’alias’ table in the [PLC] section of the matplc.conf file).

Limitations and extensions

The IEC standard allows compilers to accept extensions as long as they are within pragmas, which for the IEC 61131-3 is any text delimited by ’{’ and ’}’. We have therefore extended the accepted syntax to allow inclusion of files from within other files, by using the following syntax:
{#include "<filename>"}
where <filename> should be replaced by the name of the file to be included. File inclusion is only allowed outside the definition of IEC 61131-3 Program Organization Units (i.e. outside Funtions, Funtion Blocks, Programs, Configurations).
Our compiler also allows that C or C++ code be inserted inlined with ST or IL code. This is done by simply inserting any C or C++ code, within the two ’{’ and ’}’ delimiters, in ST or IL code. Note that this support is prelimenary and its syntax may change without notice (as everything else in this project, really...;-])
As stated previously, the IEC compiler is still at an embrionic stage, and does not yet support the full syntax and semantics of the IEC IL/ST languages.

Namely, the following elementary data types are not yet supported: TIME, DATE, TIME_OF_DAY, DATE_AND_TIME, STRING, WSTRING

In addition, the following derived data types are also not yet supported: SUBRANGE. ENUMERATION and ARRAY. Note that STRUCTREs are supported.

Standard functions and function blocks have not yet been written, except for tentative test functions/function blocks such as: BCD_TO_INT, INT_TO_BDC, SR, RS, CU, CD, CUD, R_TRIG, F_TRIG.

Due to the way located variables are implemented, we do not yet support the passing of EXTERNAL variables, declared in programs or functions blocks, as OUT or IN_OUT parameters to functions.

Another restriction is in the calling of function blocks using the formal syntax from within IL code. In this case, a parameter may not be given the value obtained from an embedded list of IL instructions. For example, consider the following function block call:

```
CAL fb_instance (      param1 = var1,      param2 = 45,      param3 = (              LD var2              ADD var3              ),      param4 = var4    )
```

In the above code sample, the embeded IL for ’param3’ is not yet supported.

[p 58] [p 1] [p 72]

$Date: 2005/06/09 18:07:00 $

[p 68] [p 1] [p 76]

MAT IL language

Introduction

The MAT IL is a simple text transcription of the traditional "stepladder" style of programming (sometimes also called "mnemonics").

The stepladder is described a rung at a time from left to right.

At the top of the scan, inputs are read from the MatPLC core. The program is then scanned an instruction at a time from top to bottom (unless a `JMP` instruction changes this). Once scanning reaches the bottom of the program or the special instruction `END`, outputs are written to the MatPLC core and the process starts over.

For all non-latching relays, make sure an `OUT` or `OUTI` instruction is scanned on every scan. Otherwise relays are latching: unless an `OUT`, `OUTI`, `SET` or `RST` instruction changes them, they remain as they were.

Instructions

`LD` *contact/register*
: Start new rung with a normal contact or a register.

`LDI` *contact*
: Start new rung with an inverted contact.

`K` *value*
: Start new rung with the given value, which can be `on`, `off` or a number.

`AND` *contact*
: Add a contact on the end of the current rung (in series).

`ANI` *contact*
: Add an inverted contact on the end of the current rung (in series).

`OR` *contact*
: Add a contact in parallel with the current rung so far.

`ORI` *contact*
: Add an inverted contact in parallel with the current rung so far.

`OUT` *coil/register*
: Connect a relay coil to the rung. If the current rung is **on**, the relay turns on, if the current rung is **off**, the relay turns off. For numeric rungs, the value is written to the register.

  There should not normally be more than one `OUT` or `OUTI` for each coil (unless one or the other is skipped using a `JMP`). Having more than one is valid, but will mean that some rungs (between the first and second `OUT`) will use the value from the first `OUT`, while a different value (from the second `OUT`) will actually appear on the output. This effect can be quite useful, of course, as long as scan-order is taken into account.

`OUTI` *coil*
: Connect an inverted relay coil to the rung. If the current rung is **on**, the relay turns off, if the current rung is **off**, the relay turns on.

  Please see the remarks at the `OUT` instruction.

`SET` *coil*
: Connect the ‘set’ coil of the relay to the rung. If the current rung is **on**, the relay turns on, otherwise no effect.

  It is somewhat more sensible to have a `SET` or `RST` in combination with an `OUT` than it is to have two `OUT` instructions. For instance, an exception logic may wish to override a previous `OUT`, setting the relay on or off as appropriate. Nevertheless, care should be taken to avoid confusion.

`RST` *coil*
: Connect the ‘reset’ coil of the relay to the rung. If the current rung is **on**, the relay turns off, otherwise no effect.

  Please see the remarks at the `SET` instruction.

`LT` *register*
`LE` *register*
`GT` *register*
`GE` *register*
: Compare the current rung with the register numerically. The result will be **on** if the comparison is true, and **off** if it is false.

`MCS`
: Master Control start - the current rung becomes a new rail. All subsequent `LD`, `LDI` and `OR` instructions will use this rail until another `MCS` or a `MCE`.

  Warning: this instruction is not a substitute for hard-wired safety and emergency stop circuits.

`MCE`
: Master Control end - cancels all current `MCS`.

`ANB`
: And block - takes the current and previous rungs and connects them in series. This also converts them into a single rung for the purposes of subsequent ANBs and ORBs (up to 7 previous rungs may be

used).

Unless you know what you are doing, you probably shouldn't have had an OUT on the previous rung.

ORB
: Or block - takes the current and previous rungs and connects them in parallel. This also converts them into a single rung for the purposes of subsequent ANBs and ORBs (up to 7 previous rungs may be used).

  Unless you know what you are doing, you probably shouldn't have had an OUT on the previous rung.

LTB
LEB
GTB
GEB
: Comparison block - takes the current and previous rungs and compares them numerically. This also converts them into a single rung for the purposes of subsequent ANBs and ORBs, which will be **on** if the comparison is true, and **off** if it is false. (Up to 7 previous rungs may be used at any time.)

END
: In main program, end program scan. In a subroutine, return. This can be used at the end of the program/subroutine, or in conjunction with JMP and LBL.

JMP *label*
: If the current rung is **on**, transfer scanning to a different part of the program. Jumps can be made in any direction, to any point in the program, but not into or out of a subroutine.

  It is valid to jump into the middle of a rung: before the jump the current rung (which is **on**) is discarded and the previous rung reinstated as the current rung (and so on).

  If the current rung is **off**, JMP has no effect and scanning continues as normal on the current rung.

LBL *label*
: Mark the location to which a JMP transfers scanning. When encountered, has no effect.

JSR *subname*
: If the current rung is **on**, transfer scanning to a subroutine. When the subroutine finishes, scanning will resume at this point in the program.

  Before the jump the current rung (which is **on**) is discarded and the previous rung reinstated as the current rung (and so on). This can be used to pass parameters to the subroutine.

  If the current rung is **off**, JSR has no effect and scanning continues as normal on the current rung.

  It is valid for a subroutine to JSR another, or even itself (directly or indirectly). In the latter case, care should be taken to avoid infinite recursion and ensure that the subroutine eventually returns.

SUB *subname*
: Marks the beginning of a subroutine. The subroutine extends to the next SUB instruction or to the end of the program (though an END can be used to return from it prematurely).

The SUB instruction always marks the end of the previous subroutine or the main program. If it is encountered in normal scan, it causes a subroutine return (in a subroutine) or a program scan end (in main program), just like an END.

The only way to get into a subroutine is with a JSR. It is not valid to JMP in and out of subroutines or between different subroutines.

RET
: If the current rung is **on**, return (in a subroutine) or end program scan (in main program).

  When this instruction results in a return from subroutine, the current rung (which is **on**) is discarded and the previous rung reinstated as the current rung (and so on). This can be used to return values from subroutines.

  If the current rung is **off**, RET has no effect and scanning continues as normal on the current rung.

POP
: Discard the current rung and reinstate the previous rung as the current rung (and so on up to 7 rungs back).

NOP
: No effect. May appear anywhere in the program.

Instructions appear one to a line. The instructions may appear in upper or lower case (ie "LD" is the same as "ld") but the contacts, coils and labels must agree exactly (ie "X1" is not the same as "x1").

Comments must appear on their own lines. They are indicated with a hash (#) or semicolon (;) at the beginning of a line.

Both instructions and comments may be indented from the left by any number of spaces and any number of spaces or tabs may be used to separate an instruction and its contact, coil or label.

At the start of each scan, both the current rung and previous rungs have undefined values (which means they could be anything). This means that the first instruction (other than LBL or NOP) must be LD or LDI and similarly a sufficient number of LD or LDI instructions must be scanned before an ANB or an ORB.

[p 68] [p 1] [p 76]

$Date: 2004/12/28 05:32:11 $

[p 72] [p 1] [p 77]

PLC5 Emulator

The manual for this module is not yet written.

Introduction

This module is an emulator of the programming language of the PLC5 line.

Instructions

[p 72] [p 1] [p 77]

$Date: 2004/12/28 05:32:11 $

OROCOS

Not yet available

OROCOS is a project for Robot Control; more information is available from its website.

Integration between MatPLC and the OROCOS project is currently in the planning stage, inching towards initial implementation.

[p 76] [p 1] [p 78]

$Date: 2004/12/28 05:32:11 $

[p 77] [p 1] [p 88]

HMI GTK

[screenshot] Preamble

This module is (c) 2001 Juan Carlos Orozco and is distributed under the GPL license.

Introduction

The purpose of this module is to build a GUI (operator screen) to read and write MatPLC points.

The Glade program is used to build the screen.

In a few words, the HMI GTK module works as follows: Glade generates a `.glade` file (an XML file representing the GUI). HMI GTK module reads and interprets this `.glade` file and uses a naming convention to connect widgets with the MatPLC points.

Naming of Widgets

The widgets are connected to the MatPLC points by giving them special names. The easiest way to construct these names is using the `widgetnamer` wizard.

Using widgetnamer

When `widgetnamer` starts, it will pop up a dialog box. Fill in the details as follows:

`point`	The MATPLC point that should be connected to this widget.
`widget number`	Normally, this will be 0. If you want several widgets showing the same point, you need to number them 0, 1, 2 etc
`type`	The type of point to which we are connecting. For coils, this will be bool (on/off), for registers you have to check what type they are and pick appropriately. All the points used by DSP are f32 (floating-point number).
`parameter 1`	This has different meanings depending on the type. For instance, for the bool type this will be the ON text or image.
`parameter 2`	This has different meanings depending on the type. For instance, for the bool type this will be the OFF text or image.

Note that for image names, no extension is required - a .xpm is added automatically.

Once you have filled it in, click the "Select" button (or press Enter in one of the text fields), then use the middle button to paste the name into glade. The entries can be cleared with the "Clear" button, or you can simply edit them to get the next widget name.

Explanation of the widget names

The names are put together as follows:
`_pointname[.number[.type][.parameter1[.parameter2]]]`

The `number` field is needed because every widget in a window must have a unique name. `type` can be bool, i32, i16, i8, u32, u16, u8 or f32. The default type is bool.

Example: `_left.0.bool.on.off`

This can be the name of a LabelWidget that will be displaying "on" or "off" depending on the state of the point named "left" at the matplc.conf file.

Kinds of widgets

The widgets that are currently supported are:

GtkLabel (Output)
: Supports all types.
: param1 = on text message.
: param2 = off text message.

GtkProgressBar (Output)
: Supports f32.
: Range is limited by the widget settings.

GnomePixmap (Output)
: Supports all types.
: param1 = on bitmap filename (.xpm extension is automatically added)
: param2 = off bitmap filename (.xpm extension is automatically added)

GtkPlotCanvas (Output)
: Supports f32.
: Additional information: see below

GtkToggleButton (Input)
: Supports bool type.
: Add Signal: toggled, Handler: update_value

GtkRadioButton (Input)
: Supports bool type.
: Add Signal: toggled, Handler: update_value
: Set the same Group-ID (eg. radio) to all connected Radio-Button’s.

GtkCheckButton (Input)
: Supports bool type.
: Add Signal: toggled, Handler: update_value

GtkButton (Input)
Supports bool type.
Add signal: pressed, Handler: update_value
Add signal: released, Handler: reset_value
Note: One can make special set or reset buttons by just adding the pressed or released signal respectively.

GtkEntry (Input)
Supports all types.
Add Signal: activate or changed, Handler: update_value

GtkHScale and GtkVScale (Input)
Supports f32, u32, u16, u8, i32, i16 and i8.
Add Signal: button-release-event, Handler: update_value

GtkSpinButton (Input)
Supports f32, u32, u16, u8, i32, i16 and i8.
Add Signal: changed, Handler: update_value

GtkOptionMenu (Input)
Supports i32.
Add only one dummy-Item to the Itemlist eg. "---".

GtkSocket (External)
Supports u32.
Point gets XID of the socket window.
With this XID you can plug from an external module.

Please take a look to demo/widgets_gtk how to use the widgets.

Odds and ends

In order for input widgets to work one must add a signal with handler = update_value.

In order to be able to write variables, they should be assigned to this module in the matplc.conf file. The default module name for the HMI module is hmi_gtk.

One can place other widgets and they won't interfere with MatPLC as long as their names don't start with an underscore "_".

Putting it all together

The suggested way of building a project is to create a project in glade named hmi_gtk. Then add a Gnome Application Window found on the Gnome palette. Delete unwanted menu options and toolbar options. The about menu option will automatically display this project credits. One can add widgets to this window and connect widgets to the MatPLC by following the previously described naming convention.

Closing main window with delete_event (from window-manager, e.g. X-Button):
The main window (app1) must connect the signal "delete_event" with the quit_handler. The quit_handler (when called by "delete_event") sets the point "quit_app1".
In matplc.conf you have to

1. create the point "quit_app1"
2. assign the point "quit_app1" to plcshutdown.

To build a multi window project one can then add normal Window Widgets found on the GTK+ basic palette of Glade. To open this additional windows we need to link a signal from a button, toolbar button or menu option. There are two ways to link windows to widgets (i.e. button):

One way is to use the 10 predefined names for windows, window1 to window10, each one with a predefined Handler function run_window1 to run_window10. When opening a window from a menu option this is the only way to call windows one just need to add the corresponding Handler name to the menu option (i.e. Handler: run_window5). This limits the number of additional windows to call from the menu options to this 10 predefined ones.

There is an alternative way of calling a window that can be used by other Widgets like the button widget. This method uses a common handler called run_window, this handler expects a window name in the Data parameter of the signal. This method does not impose a limit on the number of windows that the project can open. One could also call the predefined windows from this widgets by adding the window handler to the corresponding signal.

Example of connecting a button widget to a window.

- In the Signals tab of the button properties window write:

  Signal: clicked

  Handler: run_window

  Data: WindowName (This has to be the name of the window in Glade)

- Then type Add.

Because of performance concerns, there is a feature in the program that only allows one version of each window to be open at one time.

FIXME: How to close windows? `ikeya` wrote on the mailing list: "... *should use delet_event call back.*"

When saving the project from glade we will get a hmi_gtk.glade file, this file is what the HMI_GTK module uses to run the graphical HMI.

GnomePixmap Widget

With the GnomePixmap-widget you can display pictures in xpm-format according to the value of the associated plc-point.

There are two ways to pass the xpm-file filenames to hmi_gtk.

1. As param1 and param2 of the widget-name.

   e.g. _abcd.0.bool.on.off

   plc-point = abcd, type = bool
   This will result in the behaviour:

PLC-Point(s)	Value	Displayed Pixmap
abcd	== 0	off.xpm
abcd	> 0	on.xpm

2. Via definitions in matplc.conf under section [hmi_gtk]:

   Widget-name:

   e.g. _abcd.0.i32

   plc-point = abcd, type = i32

```
[hmi_gtk]
GnomePixmap widgetpattern pixmap1 pixmap2 ...
```

   The widgetpattern can be the name of the widget (without _) or a namepattern with wildcards (similar to filename-match) included (e.g. LED* -> all widgets starting with LED). Note: The last pattern match is used. So the order is important. If you have one widget LEDerrorXYZ and many widgets starting with LED you can do the job like this:

```
[hmi_gtk]
#           widgetpattern   value 0     1      2      3
GnomePixmap LED*                grey  green  yellow  red
GnomePixmap LEDerrorXYZ          off  error
```

   This will result in the behaviour:

PLC-Point(s)	Value	Displayed Pixmap
LEDerrorXYZ	== 0	off.xpm
LEDerrorXYZ	> 0	error.xpm
LED*	== 0	grey.xpm
LED*	== 1	green.xpm
LED*	== 2	yellow.xpm
LED*	== 3	red.xpm

GtkLabel Widget

With the GtkLabel-widget you can display the value of the associated plc-point.

There are two ways to define the format:

1. the format is chosen automatically according to the type of the plc-point.
2. Via definitions in matplc.conf under section [hmi_gtk]:

```
[hmi_gtk]
GtkLabel widgetpattern "format"
```

The widgetpattern can be the name of the widget (without _) or a namepattern with wildcards (similar to filename-match) included (e.g. abc* -> all widgets starting with abc). Note: The last pattern match is used. So the order is important. If you have one widget beltSpeed and many widgets ending with Speed you can do the job like this:

```
[hmi_gtk]
#           widgetpattern   format
GtkLabel  "*Speed"        "%5.2f km/h"
GtkLabel  beltSpeed       "%7.3f m/s"
```

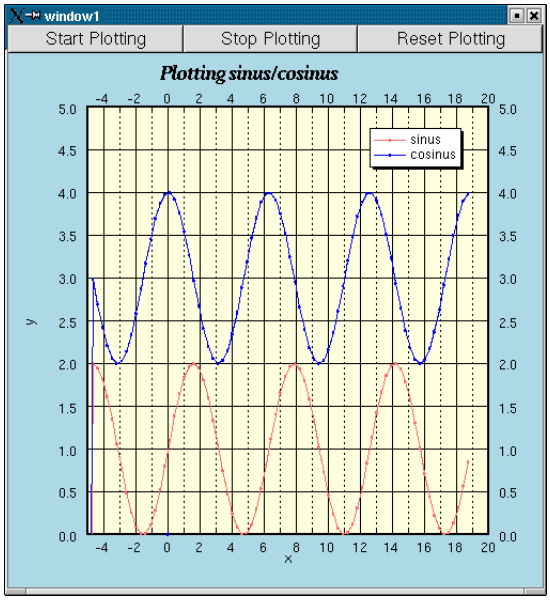
Note:

If the widgetpattern starts with "*" you have to enclose it in "".

For valid format-definitions see the printf() documentation.

GtkPlotCanvas Widget

With the GtkPlotCanvas-widget you can draw on an plotting field in two modes.



How to create:

1. Add a new "custom widget".
2. Widgetname as before with leading underline (e.g. _name)
3. Creation Function for the "custom widget" is "hmi_GtkPlotCanvas".

Definitions in matplc.conf under section [hmi_gtk]:

```
[hmi_gtk]
#-------------------------------------------------------------------------------------------
#               - command -     - param 1 -     - param 2 -     - param 3 -     - param 4 -
#-------------------------------------------------------------------------------------------
widgetname      plotctrl        startpoint      stoppoint       resetpoint
widgetname      channel         y1point         y2point
widgetname      mode            timer/plotxy

# in plotxy-mode required
widgetname      plotx           xpoint

#[optional] Default-Values see below
widgetname      widget          xsize           ysize
widgetname      title           title           x-axis          y-axis
widgetname      axis            xmin            xmax            ymin            ymax
widgetname      timer           scantime

widgetname      color           bg              plot_bg         legend_bg
widgetname      channelcolor    red
widgetname      symboltype      star
widgetname      symbolstyle     opaque
widgetname      symbolsize      2
widgetname      linestyle       solid
widgetname      linewidth       1
widgetname      pointconnect    straight
```

Explanation of the commands

[REQUIRED]:

- plotctrl:
  1. The point which will start plotting
  2. The point which will stop plotting
  3. The point which will reset plotting
- channel: Channel points limited to 20. Name of the point will be printed in legend.
- mode: current there are two modes
  1. timer-mode: The channels will be scanned every scantime (set by timer-command). The x-coordinate will be the scantime.
  2. plotxy: You have to set x-/y-coordinates. Every time the x-coordinate changes the next points (all channels) will be drawn. You can plot the points in any direction.

[REQUIRED (plotxy-mode)]:

- plotx: x-point in plotxy-mode.

[OPTIONAL]:

- widget:
  1. Horizontal size of the widget in points (Default: 600)
  2. Vertical size of the widget in points (Default: 600)
- title:
  1. Title of the plot (Default: "Title")
  2. Titling of the x-axis (Default: "x")
  3. Titling of the y-axis (Default: "y")
- axis:
  1. Startvalue for xmin (Default: 0)
  2. Startvalue for xmax (Default: 20)
  3. Startvalue for ymin (Default: 0)
  4. Startvalue for ymax (Default: 20)
- timer: Scantime in sec (timer-mode) (Default: 0.5)
- color:
  1. Background color of the widget (Default: "light blue")
  2. Background color of the plotfield (Default: "lightyellow")
  3. Background color of the legend (Default: "white")
- channelcolor:
  Color of the channel (Default: "red" for all channels) You can specify a color for each channel.
- symboltype:
  Symbol type of each channel (Default: "star")
  Possible entries: "none", "square", "circle", "up_triangle", "down_triangle", "right_triangle", "left_triangle", "diamond", "plus", "cross", "star", "dot","impulse"

- symbolstyle:
  Symbol style of each channel (Default: "opaque")
  Possible entries: "empty", "filled", "opaque"
- symbolsize:
  Symbol size of each channel in points (Default: 2)
- linestyle:
  Line style of each channel (Default: "solid")
  Possible entries: "none", "solid", "dotted", "dashed", "dot_dash", "dot_dot_dash", "dot_dash_dash"
- linewidth:
  Line width of each channel in points (Default: 1)
- pointconnect:
  Connection of the points of each channel (Default: "straight")
  Possible entries: "none", "straight", "spline", "hv_step", "vh_step", "middle_step"

GtkOptionMenu Widget

With the GtkOptionMenu-widget you can choose one item from a list of items.
The Itemlist and the associated points will be defined in the matplc.conf.
Please refer to the demo/widget_gtk example on how to use this widget.
Definitions in matplc.conf:

```
[plc]
point widgetname "" hmi_gtk i32
point p1         "" hmi_gtk i32
point p2         "" hmi_gtk i32

[hmi_gtk]
widgetname p1 "item1" "item2" ...
widgetname p2 "item11" "item12" ...
```

widgetname: Name of the widget as defined in "glade" without leading undersore.
p1...pn: Points which will be set by hmi with the corresponding value.
If you select item "item1" p1 will be set to 1, p2 is set to 0.
If you select item "item2" p1 will be set to 2, p2 is set to 0.
If you select item "item11" p2 will be set to 1, p1 is set to 0.
If you select item "item12" p2 will be set to 2, p1 is set to 0.
If the dummy-item (on top of the List) is selected all points are set to 0.
Example:

```
[plc]
point  menu1    "Menu 1" hmi_gtk i32
point  p1       " " hmi_gtk i32
point  p2       " " hmi_gtk i32

[hmi_gtk]
# widget   point  value: 1        2             3
menu1      p1     "choice1 p1" "choice2 p1"  "choice3 p1"
menu1      p2     "choice1 p2" "choice2 p2"  "choice3 p2"
```

GtkSocket Widget

Together with GtkPlug, GtkSocket provides the ability to embed widgets from your own module into the hmi_gtk module in a fashion that is transparent to the user.

How to create:

1. Add a new "custom widget".
2. Widgetname as before with leading underline (e.g. _name)
3. Creation Function for the "custom widget" is "hmi_GtkSocket".

The hmi_gtk passes the XID of that widget to the given point.
In your module you have to plug to the GtkSocket with the XID and then you can insert everything you want into the plugged window.
Please refer to the demo/widget_gtk example on how to use this widget.

WARNING:

Due to a bug in glade 0.6.2 and earlier versions, the GnomePixmap scaled state is not saved. This will cause the *.glade configuration file to loose this information each time a change is saved. In this demo the following line was inserted manually for each scaled widget: `<scaled>True</scaled>`

This simple bug makes working with scaled GnomePixmap images uncomfortable. From glade version 0.6.3 this bug was corrected, this new version could be downloaded from:
ftp://ftp.gnome.org/pub/GNOME/stable/sources/glade/glade-0.6.3.tar.bz2
ftp://ftp.gnome.org/pub/GNOME/stable/sources/glade/glade-0.6.3.tar.gz

Libglade does not recognise the scaled state either, therefore an interim solution was implemented in hmi_gtk.c module. The inconvenience is that the name of the pixmap has to be placed on the name of the widget using the following format: `.filename.ext_id`

The widget is actually scaled to the size of the widget and not using the scale factors of the GnomePixbuf.

[p 77] [p 1] [p 88]

$Date: 2005/10/11 13:35:45 $

[p 78] [p 1] [p 89]

vitrine

Vitrine is a simple text-based HMI. It is display-only (it can be paired with the `kbd` module to provide both display and control, or with buttons wired to inputs).

It is useful in small systems for which the overhead of graphics would be excessive.

Config

There are three settings:

`background=`*`filename`*
: This specifies a plain text file to be loaded as the unchanging part of the screen.

`show` *`point row col [type] [on [off]]`*
: This table specifies the points to be shown on the screen, one point per row. The columns are:

  `point`
  : The MatPLC point to be shown.

  `row`
  : The row on the screen to use.

  `col`
  : The column on the screen to use.

  `type`
  : (optional, default `bool`) The type to be shown: `f32 | i32 | u32 | i16 | u16 | i8 | u8 | bool`. The default is `bool`, which is used for on-off points (coils, discrete inputs).

  `on`
  : (optional, default "1") The string to use when the point is on. Only valid for the default `bool` type. In bespin (and CVS before June 2005), only the first character is shown on the display.

  `off`
  : (optional, default "0") The string to use when the point is off. Only valid for the default `bool` type. If you want a string which only appears when the point is ON and disappears when the point is OFF, use `""` as the *`off`* marker (two double-quote marks). In bespin (and CVS before June 2005), only the first character is shown on the display, and a blank must be written as `" "` (double-quote, space, double-quote).

`graph ...`
: **FIXME** Document how to do text-based graphs. See `demo/basic_dsp/vitrine.conf` for an example.

[p 78] [p 1] [p 89]

$Date: 2005/05/26 11:20:33 $

[p 88] [p 1] [p 91]

Data Logger

Introduction

There are two logger modules mat/io/logger/logger and mat/io/logger/logger_db
logger module logs data to a flat file.
logger_db module logs data to a MYSQL database.

Logger module configuration

Configuration options that need to be set to use a logger module:
file = *filename*
points *pointname1 pointname2 ...*
scan_period = *sec*
Sample:
[LOGGER]
file = chaser.log
points L1 L2
points L3 L4
scan_period = 0.25

Logger_db module configuration

Configuration options that need to be set to use a logger_db module:
host = *host name*
user = *user name*
password = *password*
#port_num = *port#*
#socket_name = *socket name*
db_name = *database name*
table = *table name*
points *pointname1 pointname2 ...*
scan_period = *sec*
Sample:
[LOGGER_DB]
host = localhost
user = matuser
password = mat
#port_num = 0
#socket_name =
db_name = matlog

table = demolog
points L1 L2
points L3 L4

Data logger demo

To try these modules use the /mat/demo/basic_logger module To use the logger_db module you should have a MYSQL connection available with the database already created. The table is automaticaly created if it does not exist. If a table with the same name exist in the database but the structure is not compatible then the logger_db will not log, but if a table with the same name exist and the structure is compatible then logger_db module will append rows to this table. Also there is a commented line in /mat/demo/basic_logger/matplc.conf that needs to be uncommented to test the logger_db module. This is because this sample uses only the logger module by default.

[p 88] [p 1] [p 91]

$Date: 2004/12/28 05:32:11 $

[p 89] [p 1] [p 92]

hmiViewerServer

This module is Copyright © 2001 Juan Carlos Orozco and is distributed under the GPL license [p 165] .

Introduction

hmiViewerServer.py is a multithreaded socket server that is used to comunicate MatPLC to one or more hmiViewer applets from the VISUAL (visual.sourceforge.net) project.

Description

One of the main advantages of this module is that the points for communication are not defined in this module, it gets the point names dynamically from the hmiViewer applet and then tries to connect them to MATplc. In case there is a point that is not defined in MATplc the server still keeps a local copy of this point with its value so the hmiViewer applet will still be able to read and write this point but it will not be accessible from the rest of the MATplc modules.

[p 89] [p 1] [p 92]

$Date: 2004/12/28 05:32:12 $

[p 91] [p 1] [p 94]

Common I/O options

Introduction

Most I/O modules use a common syntax for mapping physical I/O points to MAT points. In addition, each I/O module may have additional configuration.

Mapping

The mapping between the physical I/O and the plc points is specified using the "map" table. The format of each line is

```
map [inv | invert] {in | out} <io_addr> <matplc point>
```

`inv` or `invert`
: invert the value being read from/writen to physical IO All the bits are inverted, which is handy for physical I/O that uses negative logic (on = 0, off = 1) - whether it's simply a wrongly wired switch, or a sensor that's only available N/C where previously it was N/O.

`in`
: copy the state of the physical Input/Output to the matplc point (it is an input)

`out`
: copy the state of the matplc point to the physical Output (it is an output)

`<io_addr>`
: address of physical input/output. The acceptable format this address will depend on the type physical I/O we will be interfacing with. Each different MatPLC I/O module accepts a different io_addr format.

`<matplc point>`
: the MAT point to which this physical input/output is to be mapped.

Example

For instance, suppose that we are using the parallel port I/O module. We have leds that will light up when the output is low, so we choose to invert all the outputs so the leds will light up when the plc point is set to 1.

D.x is the acceptable format for the parallel port I/O module. D specifies the D register, and x the bit of that register.

The mapping table might look something like this:

```
map inv out D.0 L1
map inv out D.1 L2
map inv out D.2 L3
map inv out D.3 L4
```

[p 91] [p 1] [p 94]

$Date: 2004/12/28 05:32:11 $

[p 92] [p 1] [p 95]

ABEL I/O

The abel module communicates with the ABEL library for communication with AB PLC5 and similar. The library (but not the module) was written by Ron Gage.

Warning: the module has never been tested.

Its configuration is quite simple. Apart from the usual mapping table, it only has one parameter, `IP`, which gives the IP address of the PLC to be contacted.

In the mapping table, standard AB addresses are used, as supported by the ABEL library.

[p 92] [p 1] [p 95]

$Date: 2004/12/28 05:32:10 $

[p 94] [p 1] [p 97]

CIF I/O

Introduction

This MAT module, together with the corresponding kernel module (device driver), interfaces with CIF I/O cards.

These cards support the following networks and protocols (in alphabetical order): ASi, CANopen, ControlNet, DeviceNet, Interbus, ModBus Plus, PROFIBUS and Sercos. Hilscher implements the protocol and pays the license in their firmware. By using this as the interface to the bus, we can still use GPL [p 165] -able code to talk to the firmware on the card and out through the network.

Download note

The CIF kernel driver is not included in some of the downloads, for instance the weekly tarball, because of its size. If you plan to use this module, make sure that you download a version which includes it.

Mapping

The <io_addr> column of the map is a simple *offset.bit* format. The *offset* ranges from 0 to the process image size, and the *bit* ranges from 0 to 7.

Numbers may be given in decimal (58) or hexadecimal (0x3A). The *.bit* part may be omitted.

MatPLC points larger than 1 bit will be mapped starting at the given <offset>.<bit> location, and carry on from there. Note that this means that the data of a matplc point may also fall onto the subsequent bytes of the process image, if required.

Examples:

`map out 0.0 a_mat_1bit_point`
: the mat point will only fall on 0.0

`map out 1.0 a_mat_16bit_point`
: the mat point will fall on 1.0 to 2.7

`map out 3.0 a_mat_32bit_point`
: the mat point will fall on 3.0 to 6.7

`map out 10.4 a_mat_8bit_point`
: the mat point will fall on 10.4 to 11.3

`map out 20.6 a_mat_32bit_point`
the mat point will fall on 20.6 to 24.5

Configuration

`BoardID`
The CIF card BoardId. This is the ID the kernel module uses to identify the card we want to use.

The CIF module of the MatPLC does not directly access the CIF card. It does so through a kernel module (i.e. a device driver) specific for CIF cards. When this module is loaded, it finds every PCI based CIF card automatically. Cards on the ISA bus have to be explicitly specified on the command line when the device driver is loaded. Each card that the device driver can correctly access is given a unique number from 0..3. Since this module uses the device driver to control the CIF card, what the module needs is the number the device driver gave the CIF card.

Defaults to 0

`DPMsize`
The DPM (Dual Port Memory) Size, in kBytes.

This is the size of the DPM on the card the module will be using. Actually this parameter is not very important, this is basically just to allow the CIF mofule to verify at configuration time if any plc point will be mapped onto an offset that falls outside the card's process image. The In and Out process images are disjoint, and do not overlap. The size (in bytes) of each process image depends on the DPM size (in kBytes) and is given by: ((DPMsize * 1024) - 1024) / 2

At runtime the module will verify if the card that it is using really does have a DPM the size the user says it should have.

Defaults to 8. Other possible value seems to be 2 (kBytes).

`timeout`
Maximum timeout when trying to access the CIF card. Value is interpreted as ms, 0 means no timeout. Default is 100 (ms).

[p 94] [p 1] [p 97]

$Date: 2004/12/28 05:32:10 $

[p 95] [p 1] [p 100]

comedi I/O

The comedi module uses the comedi library to interface with various I/O cards (almost 250 as of this writing, from various manufacturers). The library (but not the module) was written by the comedi project.

Example howto

There is a howto in the hardware compatibility section of the webpage, outlining how to use the Advantech PCL-812PG card with this module.

Introduction

Setting up the hardware is described in Chapter 2 of the comedilib manual. Various comedi utilities and sample programs will let you install the device driver, verify that it was correctly detected, obtain information about the input and output ranges of the various devices, and so on.

In comedi, the inputs and outputs on each device (card) are grouped into subdevices. Each input or output is then referred to as a channel. All three of these (devices, subdevices and channels) are numbered, starting from 0. In addition, analogue channels can have multiple ranges and references, if the card supports these, and they can be obtained raw (as an integer) or converted to physical units (volts or milliamps).

Addresses

In the mapping table, comedi addresses are specified as follows:

Examples

`I2.3`
: digital input, device 0, subdevice 2, channel 3.

`1:X7.3/2_DIFF`
: converted analog input, device 1, subdevice 7, channel 3, range 2, reference AREF_DIFF.

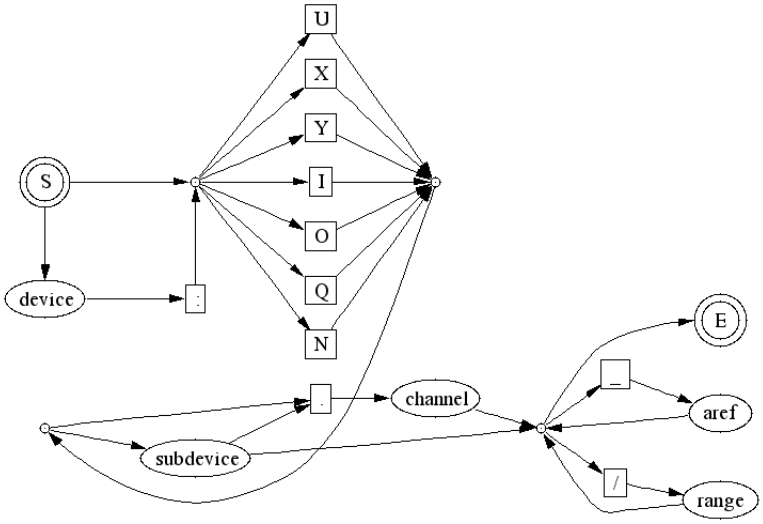
`U.2`
: analog output, device 0, subdevice 0, channel 2, default range and reference.

Format

Each address has the form (some parts optional):

```
device: type subdevice.channel options
```

This diagram shows which parts are optional, which are mandatory, and the order in which they may be used. To make a valid address, start in the double circle labelled **S**, and follow the arrows until you get to **E**. Boxes denote actual letters and punctuation to use, ovals the various settings, as explained below. The small circles are simply junctions.



Details

device:
: either the device filename, or the number of the device ("`/dev/comedi`" will be prepended in the latter case). The device specification may be omitted, including the colon, in which case 0 will be used.

type
: denotes the direction, kind and conversion to physical units; it is mandatory. Lowercase letters may also be used, which is probably a good idea for `I` and `O` (so they don't get mistaken for numbers). The floating-point format used by the `X` and `Y` types is the same as used by the DSP module [p 58] .

type	meaning	bits	format
`I`	digital input	1	on/off
`O` or `Q`	digital output	1	on/off
`N`	analog input (raw) (mnemonic: iN)	1-32	integer
`U`	analog output (raw) (mnemonic: oUt)	1-32	integer
`X`	analog input converted to physical units	32	float
`Y`	analog output converted from physical units	32	float

subdevice.channel
: the subdevice and channel to use on the device; one or the other may be omitted, but not both; omitted parts default to 0.
  - `1` is the same as `1.0`
  - `.1` is the same as `0.1`

options
: There are two options, either or both of which may be used for analog inputs and outputs; there are no options for digital I/O.

  `/`range
  : specifies the range to be used, as a number. Default 0.

  _ref
  : specifies the reference to be used, either as a number, or as a keyword (so `0`, `GROUND` and `GND` are all equivalent). Default `GROUND`. Mnemonic: the value at the low end; hence underscore.

[p 95] [p 1] [p 100]

$Date: 2004/12/28 05:32:10 $

[p 97] [p 1] [p 106]

Modbus I/O

Modbus master

The modbus master is actually three modules, one each for Modbus/RTU, Modbus/ASCII and Modbus/TCP. When you're running the demo, don't forget to select the correct module in the `[PLC]` section of the config.

```
[PLC]
# Uncomment the modbus protocol version you wish to run...
module modbus_m "../../io/modbus/modbus_m_rtu"
#module modbus_m "../../io/modbus/modbus_m_asc"
#module modbus_m "../../io/modbus/modbus_m_tcp"
```

Configuration

The configuration consists of three tables:

network
: connection information (serial port or TCP address)

node
: the devices connected to each network (modbus ID)

map
: the usual mapping of actual I/O to MatPLC points (registers)

Network configuration table

This specifies the connection information - serial port or TCP address, and any associated settings. There can be any number of these - one module may talk to several serial ports or several TCP slaves; on the other hand, having a separate module for each port or each TCP address will allow the slaves to be contacted simultaneously rather than one by one.

The format of the table is:
`network <network_name> <protocol> <parameters>`

`<network_name>`
: The name used to refer to this network in the rest of this config.

`<protocol>`
: {rtu | ascii | tcp}
This must agree with the module being used.

`<parameters>`
: These depend on the type of network:

For rtu and ascii networks

```
[device <filename>]
[baudrate <x>]
[parity {even|odd|none}]
[data_bits <x>]
[stop_bits <x>]
[ignore_echo {true|false}]
[timeout <x>]
[send_retries <x>]
```

For tcp networks

`[host <x>]`
: the IP address or DNS name of the slave

`[port <x>]`
: x is either the port number on which the slave is listening, or the service name, which will be mapped onto the port number according the configuration in the /etc/services file (**FIXME** - what is the default?)

`[timeout <x>]`
: (**FIXME** - what does this do? what is the default?)

`[send_retries <x>]`
: (**FIXME** - what does this do? what is the default?)

`[TCP_close {true | false}]`
: true
  : (default) Close the connection after each scan. This is slightly slower, as the connection must be re-established at the beginning of the next scan.

  false
  : Keep the connection open between scans. However, some slaves may only support a limited number of simultaneous masters, and this will keep one of those limited slots occupied for as long as this module is running, potentially blocking out other masters.

  The openmodbus specification sugests that all TCP connections should be closed if they are going to stay idle for longer than 1 second, so that you should set `TCP_close` to `true` if the scan time for this module is longer than a second, and `false` if it's shorter. In practice, the important consideration will be how many masters are accessing this slave, compared with the slave's "maximum simultaneous masters" specification, and the relative priority between those masters.

```
# Sample network configurations...
network rtu_net         rtu   device /dev/ttyS1 baudrate 9600 parity none data_bits 8 stop_bits 1
network asc_net         ascii device /dev/ttyS1 baudrate 9600 parity none data_bits 8 stop_bits 1
network tcp_net         tcp   host   localhost  port 502  timeout .1 send_retries 1 TCP_close tru
```

Node address table

This is a fairly simple table, giving names to all the nodes based on their Modbus ID. Each line has the form:

`node <slave_name> <network_name> <slave_addr>`

`<slave_name>`
: The name used to refer to this slave node in the rest of the config.

`<network_name>`
: The name of the network, as declared in the `network` table (above).

`<slave_addr>`
: The modbus slave ID, 0..255

```
# Sample slave configurations...
node speed_drive    tcp_net 34
node digital_IO     rtu_net 0
node analog_IO      asc_net 23
```

Mapping table

Each line has the form:
`map [inv | invert] {in | out} <slave>.<reg_type>.<reg_addr> <matplc point>`

`inv`
`invert`
: (optional) invert the value being read from/writen to physical IO

`in`
: copy from the Modbus device to the MatPLC point

`out`
: copy the state of the MatPLC point to the Modbus device

`<slave>`
: the name of the slave node, as it appears in the node address table.

`<reg_type>`
: Which register type (data table) to access, according to this table:

type	data table	long address	in/out	Modbus fn. used		Amount of data read/written
				in	out	
`in_bit`	Discrete Input	1xxxx	must be mapped `in`	0x02	-	The number of bits transfered is the same as the number of bits in the MatPLC point being mapped. The address given in the mapping is the adress of the first bit.
`out_bit`	Coil	0xxxx	may be mapped `in` or `out`	0x01	0x0F (15)	
`in_word`	Input Register	3xxxx	must be mapped `in`	0x04	-	If MatPLC point being mapped has 16 or less bits, 1 word will be trnsferred. If it has between 17 and 32, 2 words will be transferred. The address given in the mapping is the adress of the first word.
`out_word`	Holding Register	4xxxx	may be mapped `in` or `out`	0x03	0x10 (16)	

`<reg_addr>`
: 1..10000
  Note that, as per the standard, `<reg_addr>` is 1-based. If your documentation lists Modbus addresses starting with 0, you will have to add 1 to each of them.

`<matplc point>`
: the MatPLC point to map

```
# Sample IO table
map out digital_IO.out_bit.1 L1
map out digital_IO.out_bit.2 L2
map out digital_IO.out_bit.3 L3
map out digital_IO.out_bit.4 L4
```

Error logging

The first time an error occurs (timeout, received invalid response frame, received error frame, etc...) when communicating with a device, this will be logged as an error at level 2. All subsequent errors for the same device will be logged as an error at level 3. When a device comes back up, this will be logged as an error at level 2.

Modbus slave

The modbus slave module is partially implemented and has not undergone serious testing. Currently, it supports TCP/IP transport only. Furthermore, functions 0x03, 0x04, and 0x06 only are available.

Access control of MatPLC points through Modbus slave is based on the standard MatPLC architectural concept: in order to write to a point the Modbus slave must be the owner of this point.

The modbus slave module, whose default name is "modbus_s", should be enabled in the MatPLC configuration with a "module" keyword and the path to the module's executable. **Note that the ASCII AND RTU versions are not implemented yet and their instances below are for illustration only.**

```
[PLC]
# Uncomment the modbus protocol version you wish to run...
module modbus_s "../../io/modbus/modbus_s_tcp"
#module modbus_s "../../io/modbus/modbus_s_asc"
#module modbus_s "../../io/modbus/modbus_s_tru"
```

Configuration

The configuration consists of two tables:

endpoint
: connection information

map
: the mapping of actual I/O to MatPLC points (registers)

Endpoint configuration table

This configures endpoints that MatPLC will listen on for incoming requests - TCP address and any associated settings. There can be any number of these - one module may listen on several interfaces.

The format of the table is:
`endpoint <network_name> <protocol> <parameters>`

`<network_name>`
: The name used to refer to this network in the rest of this config.

`<protocol>`
: {rtu | ascii | tcp}
: This must agree with the module being used (tcp is the only option currently supported).

`<parameters>`
: These depend on the type of network:
: For tcp networks
: `[host <x>]`
: the IP address or DNS name of the interface to listen on. 0.0.0.0 stands for INADDR_ANY.
: `[port <x>]`
: x is either the port number on which the slave is listening, or the service name, which will be mapped onto the port number according the configuration in the /etc/services file

(**FIXME** - what is the default?)

`[address <x>]`
: Modbus slave address.

`[connections <x>]`
: The number of simultaneous incomming connections to accept.

`[mode <x>]`
: The mode in which the slave is run: 0 - async, 1 - sync. The default value is 0

```
# Sample endpoint configuration...
endpoint tcp_net  tcp   host 0.0.0.0 port 5502  address 34 mode 0 connections 1
```

Mapping table

The object data size is determined by the invoked modbus function. For instance, if a coil reading is requested a single bit of the mapped MatPLC point will be transmitted. Likewise, in register operations, the data object size is 16 bits. Therefore, in order to transmit an entire MatPLC point of maximum size (32 bits) two register operations are needed. Alternatively, a multiple register function may be utilized (currently not supported). Each line has the form:

`map slave <shift>.<matplc point>`

`<shift>`

The number of bits to shift a MatPLC point right before mapping it to a modbus type on reading or the number of bits to shift a modbus type right before mapping it to a MatPLC point on writing. The following aliases are available: "high" for 16 and "low" for 0.

`<matplc point>`

the MatPLC point to map

Useful links

Modbusfw - Modbus/TCP Filtering on Linux Firewalls
: A set of kernel patches for Linux Netfilter to allow firewall policy decisions (DROP, DENY, ALLOW, etc.) to be made based on Modbus/TCP header values including modbus function code, providing better access control than simply blocking port 502. However, as of Sep 2003 this is an initial prototype release and they've done only limited testing so any feedback, new feature requests, and/or help with development is appreciated. See http://modbusfw.sourceforge.net for details.

[p 97] [p 1] [p 106]

$Date: 2004/12/28 05:32:11 $

[p 100] [p 1] [p 108]

Parallel port I/O

Introduction

The parallel port has 17 usable bits of I/O, some of which can be configured either as input or as output. Up to 12 can be configured as outputs, and depending on the hardware and access method either 13 or all 17 can be configured as inputs.

The parallel port can be accessed either directly (specify the `io_addr` variable, below) or through the kernel driver (specify `dev_file`).

The bits are arranged into three registers, D, S and C. Register D can be either input or output, S is always input. The C register is usually output, but it can be used as input if the port is accessed directly (not through the kernel driver) and the hardware supports it, which varies between manufacturers. The direction of these configurable registers is inferred from the mapping; or you can specify it directly.

The format of the `<io_addr>` column for the map is `R.x` where R is the register (D, S or C) and x is the bit number. The acceptable address ranges are listed in the table.

Register	Range	Number	Direction	Note	Pins
D	D.0-D.7	8	output/input	LS TTL	2-9
S	S.3-S.7	5	input	LS TTL	15/13/12/10/11
C	C.0-C.3	4	usually output	TTL Open Collector	1/14/16/17

Note that while some of the S and C bits are inverted by the parallel port hardware, this module re-inverts all these bits to present a coherent positive logic interface. However, you may wish to verify the boot-up status of those used as outputs to avoid unexpected actions during startup.

Configuration

`io_addr`

The parallel ports base address. 0x378 is the default if no io_addr is specified. You may also like to try 0x278 if the first doesn’t work.

Example: `io_addr = 0x378`

NOTE: this is unrelated to the `<io_addr>` column of the map.

`dev_file`
: If you prefer to go though the kernel driver, then specify the device file.

  Example: `dev_file = /dev/plc_parport0`

  NOTE: If both of the above methods of accessing the parallel port are configured, then the kernel driver will be used. Only if this method fails will it fall back to the direct access mode, using the specified (or default) io base address.

`Ddir`
: defines the direction {in | out} the D register should use. If this is omitted, it will be automatically inferred from the mapping.

  Example: `Ddir = out`

`Cdir`
: defines the direction {in | out} the C register should use. If this is omitted, it will be automatically inferred from the mapping.

  Example: `Cdir = in`

  See warnings above about using the C register for input.

The S register is always input; its direction is not configurable.

Example

```
# use this device
dev_file = /dev/plc_parport0
# fallback to this hardware address
io_addr = 0x278

# Use these four bits as outputs; they’re wired to light up when the output
# is low, so we invert them here.
# This also automatically configures the D register for output.
map inv out D.0 L1
map inv out D.1 L2
map inv out D.2 L3
map inv out D.3 L4
```

[p 100] [p 1] [p 108]

$Date: 2004/12/28 05:32:11 $

[p 106] [p 1] [p 109]

Pontech sv203b

Introduction

This is a Python module that interfaces MatPLC with the sv203b servo board from Pontech.
The board uses an RS232 serial interface to comunicate with a PC using a ASCII protocol.
The module was developed as part of the MatPLC Live-Demo, it is used to move the robot arm.

Functions

Class name: sv203b
sv203b class methods:
open(port, speed, rtscts, timeout)
Opens the serial port. It needs to be called before anything else can be done with the module.
close()
Closes the serial port.
move(motor, position)
Command to move a servomotor to a certain absolute position. Motor is a number from 0 to 5.
incr(motor, position)
Same as move but using a relative position.
pos(motor)
Returns the selected servo motor current absolute position.

[p 106] [p 1] [p 109]

$Date: 2004/12/28 05:32:11 $

[p 108] [p 1] [p 110]

How to do bumpless transfers

Status: this functionality is **partially available**, as described below.

When using a PID controller, such as the one in the DSP module [p 58] , it is often useful to be able to switch between manual and automatic modes smoothly.

In the MatPLC, the two directions of switching are set up separately, so we'll cover each in turn.

By way of example, let's say we have the following points:

point	type	function
`Output`	`f32`	The output of the PID block, used to control the process.
`ManOutput`	`f32`	The setting selected manually.
`ManMode`	1-bit	Mode selector: 0=automatic, 1=manual

Manual-to-Automatic transfers

These have to be configured on the PID block. Simply add the following two parameters to the end of the pid block line:
`man_out ManOutput man_mode ManMode`

Note: bumpless transfers are not supported when the I coefficient is zero.

Automatic-to-Manual transfers

This depends on which module you use for the manual control, and how manual control is done.

Currently, the hmi_gtk module does not support bumpless automatic-to-manual transfers.

If you're using some other program for the manual control, you can ensure bumpless transfers by copying `Output` to `ManOutput`, either just before switching to manual mode or whenever you're in automatic.

If the manual control is done by "up" and "down" buttons, then it's even easier: again, in automatic mode, copy `Output` to `ManOutput`; in manual mode, just increment and decrement `ManOutput` as normal.

[p 108] [p 1] [p 110]

$Date: 2004/12/28 05:32:10 $

[p 109] [p 1] [p 112]

Introduction to custom modules

Most of the time, a combination of existing modules (generic and/or specific) will fulfil the requirements of a project. However, sometimes only a custom-made module will do. This chapter provides a short tutorial on writing custom modules.

Unless you need a custom module, you can safely skip this chapter.

Custom modules can be written in several languages. They fall into two broad groups, C and everything else.

C
: C in a group of its own because it's the native language of the MatPLC. As such, it allows access to 100% of the functionality as soon as it appears in the MatPLC, and is a little more efficient (though with current computers that generally doesn't matter).

Everything else
: Writing C is a specialist job, and sometimes it's better to write in a more user-friendly language such as python [p 113] (sort of like BASIC or VB but about three decades more modern). If you are already using one of these languages elsewhere in the project, you can also connect directly from there to the MatPLC without going through an intermediary.

  If you decide to use one of the other languages, you should still skim the chapter on C, as the other languages may be incomplete (please file a bug or ask on the mailing list if there's a function there that you want to use from one of the other languages), and in any case they generally follow the same layout and structure.

Another purpose of this chapter is to explain somewhat the workings of the supplied generic and specific modules. The modules supplied with the MatPLC work exactly the same way as custom modules. Currently they're all written in native C [p 117] .

Contributing modules to the project

If you write an interesting custom module that you think other people would find useful, you can contribute it to the project. There's no difference between writing a custom module and writing a module to contribute.

Just about the only requirement is that the module has to be under the GPL licence [p 165] . We may make some adjustments to the module to make it fit better with our intended architecture or project organization, of course, or merge it with an existing module; but if it's something that people would generally find useful, we'd love to have it in the CVS!

For other ways of contributing to the project, please see the Contributing to the MatPLC [p 146] chapter.

[p 109] [p 1] [p 112]

$Date: 2004/12/28 05:32:09 $

[p 110] [p 1] [p 113]

Custom modules - Languages other than C

Introduction

The MatPLC is written in the C language, so any new functionality will first be available in C. Also, when we add a new language, it will usually only cover the most basic functionality at first. For this reason, you should also skim the native C section [p 117] , so that you’re aware of the functionality that’s possible with the MatPLC.

If there’s a function described there that you wish to use from one of these languages, and it’s not available, please file a bug report or ask on the mailing list - it’s usually a very simple matter to add a single function or group of functions to the existing interfaces.

[p 110] [p 1] [p 113]

$Date: 2004/12/28 05:32:11 $

[p 112] [p 1] [p 115]

Custom modules - Python

Introduction

The Python language was designed to be powerful yet have very clear syntax. It has modern features like modularity, OO, exceptions, dynamic data types, and dynamic typing. For more details, visit its homepage.

User interfaces can be easily built for python scripts using tools such as `glade`.

Installing into python

The python extension must be installed before it can be used.

1. change to the `lang/python` directory
2. (optional) ./setup.py build
3. *as root:* ./setup.py install

If you don't have the MatPLC shared library installed in the usual place you'll also have to set the environment variable `LD_LIBRARY_PATH` to point to the MatPLC `lib/.libs` directory before starting python or the python script.

Example

```
import matplc

matplc.init("example")

foo = matplc.point("foo")
bar = matplc.point("bar")
es_out = matplc.point("es_out")

try:
        while 1:
                matplc.scan_beg()
                matplc.update()

                if foo.get():
                        bar.set(1)
                else:
                        bar.set(0)

                matplc.update()
                matplc.scan_end()
finally:
```

```
        es_out.set(1)
        matplc.update()

matplc.done()
```

Commentary

The first two and last two lines inside the loop are boiler-plate. Most programs will have them, since most programs will want a main loop. However, for instance GUI-oriented programs will not, and the interface supports that (ignore the scan_beg() and scan_end(), and just do an update() before reading points or after setting them).

The logic would presumably be more complicated than that, of course, but this is just a small example.

Note the use of the try/finally block - if there's any problem in the main body of the program, this will ensure that the module sets the output `es_out` (presumably connected to the ES circuit) before it exits.

Other functionality

You can use the built-in `len(foo)` function to get the length of the point `foo` in bits and `foo[0]` or `foo[0:3]` to get subpoints that access particular bit(s) of the point `foo`.

Instead of `foo.set(1)` and `foo.set(0)`, you can also use `foo.set()` and `foo.reset()` respectively.

Null points can be obtained using `matplc.point()`

Floating-point points (f32, as used by the DSP module [p 58]) can be obtained using `matplc.point_f(`*`name`*`)`

[p 112] [p 1] [p 115]

$Date: 2004/12/28 05:32:11 $

[p 113] [p 1] [p 117]

Custom modules - Tcl

Introduction

The Tcl language was designed to be a simple scripting language. Unless you're already familiar with it or have some other reason for using it, it's probably better to use python [p 113] instead.

Installing

The tcl interface has to be installed before it is used. After compiling the rest of the MatPLC, change to the `lang/tcl` directory and type `autoconf; ./configure; make` as usual, followed by `make install`

See the `README` file in the `lang/tcl` directory for more details.

Commands and options

This page describes the Tcl commands implemented by the MatPLC extension; it assumes a working knowledge of Tcl programming and the MatPLC 'C' language programming interface.

Load the module with an explicit `load` command, or with `package require lplc`.

The module implements a single new Tcl command, lplc, which has the form:

```
lplc subcommand [options...]
```

The following subcommands and options are implemented:

`lplc init [-module` *`string`*`] [-array` *`arrayname`*`] [--PLCoptions]`
: Attaches to a running MatPLC. Due to the nature of the plc_init() routine in matplc.a, this call does not return until initialization is successful.

  The default module for Tcl scripts is "TCL". This can be overridden with the -module option.

  If an array name is specified, the init command creates an associative array with the specified name, populated as follows:

`array(count)`	number of points
	name of point at index x (where 0 <= x < count)
`array(x,name)`	If the point at index x is not valid (i.e. handle.valid == 0), the name is INVALID. (this should not happen, as the indeces are dictated by the configuration.)

The --PLC options specified by the `plc_init()` routine can be passed on the command line as well. Note that this list includes `--PLCmodule=`*`string`*, another method for overriding the default module.

`lplc done`

Disconnects from the MatPLC.

`lplc getpt [-name` *`pointname`*`]`

Returns the current value of the named point.

Generates a Tcl error if a handle to the named point could not be got.

Note: the getpt subcommand performs a `plc_update()` and then a `plc_get()`

`lplc setpt [-name` *`pointname`*`] [-value` *`value`*`]`

Sets the named point to the specified value.

Generates a Tcl; error if a handle to the named point could not be got.

note: the setpt subcommand performs a `plc_set()` and then a `plc_update()`

`lplc update [-array` *`arrayname`*`]`

Updates the module's copy of the MatPLC data.

If an array name is specified, the array is populated as follows:

`array(x,value)`	value of point at index x

[p 113] [p 1] [p 117]

$Date: 2004/12/28 05:32:11 $

[p 115] [p 1] [p 118]

Introduction to custom modules in C

This chapter provides a short tutorial on writing custom modules in C.

You should probably still skim this chapter even if you're writing your module in one of the other languages, as they may be incomplete, their descriptions sometimes refer to the underlying C functions, and in any case they generally follow the same layout and structure.

Please file a bug or ask on the mailing list if there's a function in here that you want to use that's missing from the language you want to use. It's usually a very simple matter to add a single function or group of functions to the existing interfaces.

[p 115] [p 1] [p 118]

$Date: 2004/12/28 05:32:09 $

[p 117] [p 1] [p 122]

A basic custom module

As far as a module is concerned, the MatPLC is a library. It provides various functions, of which only about five or seven are the most important. The others provide things like retrieving values from the config file - useful, but not vital.

Usual usage

There are two usual ways of structuring modules: the first is the "classical scan" familiar from stepladder, while the second is a "data interface", where MatPLC and some other system exchange data.

Classical scan

Here, the behaviour resembles a stepladder scan: at the top of the loop, data is read from the inputs (and other parts of the MatPLC), then the logic is executed, and then data is written to the outputs (and other parts of the MatPLC).

This is the template for a classical-scan module:

```
#include <plc.h>

int main(int argc,char *argv[])
{
  plc_init("modulename",argc,argv);

  /* initialization goes here */

  while (1) {
    plc_scan_beg();
    plc_update();

    /* body of loop goes here */

    plc_update();
    plc_scan_end();
  }
}
```

Data interface

Here, the module does data interchange between MatPLC and something else (the "other"). In the first half of the loop, any data that’s going from the other to the MatPLC is written into MatPLC points. In the second half of the loop, data is taken from MatPLC points and sent to the other. All the I/O modules work this way, exchanging data between MatPLC and the I/O.

One advantage is that the module doesn't need to keep track of which points are read-only and which are read-write in the MatPLC; in the first half, all points may be written, and in the second half all points may be read. The `plc_update()` discards values for which the module lacks write permission.

This is the template for a data-interface module:

```
#include <plc.h>

int main(int argc,char *argv[])
{
  plc_init("modulename",argc,argv);

  /* initialization goes here */

  while (1) {
    plc_scan_beg();

    /* process data going to MatPLC from the "other" */

    plc_update();

    /* process data coming from MatPLC to the "other" */

    plc_scan_end();
  }
}
```

Filling in the template

Initialization

One of the main things you need to do here is to get handles to all the points your program will access.

```
foo = plc_pt_by_name("foo");
bar = plc_pt_by_name("bar");
```

`foo` and `bar` should be declared as variables of type `plc_pt_t`. If you want to be safe, you should also check that `foo.valid` and `bar.valid` are non-zero after the above initialization.

Throughout the rest of the program, you will then use the variables `foo` and `bar` to refer to those two points.

Body of loop

The two most important functions here:

`plc_get(`*`point`*`)`
: read point

`plc_set(`*`point`*`,`*`value`*`)`
: write the value to the point

These are used both for coils and for integer registers. When used for coils, 0 means OFF and 1 means ON.

When used for registers, value is treated as unsigned 32-bit integer (MAT defines the type `u32` for this purpose). Narrower points can also be treated as unsigned integers - for instance, a plc_get() on an 8-bit point will return a number between 0 and 255.

As usual with PLC-style logic, points are read from outside the module at the top of the loop and written to the outside at the bottom (that's what the `plc_update()` does). Last value written wins.

Remember that for `plc_set()`, the module needs point ownership in the `matplc.conf` file. There is no warning if you try to write to a point you don't own; the changes are simply discarded and the point value remains unchanged.

Processing data going to/from MatPLC

Like "Body of loop", you will use the two functions `plc_get(`*`pt_handle`*`)` and `plc_set(`*`pt_handle`*`,`*`new_value`*`)`. When processing data going **from** the MatPLC, you will only use the first of these (`plc_get()`); when processing data going **to** the MatPLC, you will only use the second (`plc_set()`).

Other than this separation, the previous section applies.

Summary of the basic functions

The five most important functions are:

`plc_init(module_name,argc,argv)`
: Initializes the library. The module_name should be a string. If argc and argv are not available, pass 0 and NULL instead.

`pt_handle = plc_pt_by_name(pt_name)`
: Obtains a handle to a MatPLC point, required for the `plc_get()` and `plc_set()` functions. The pt_name should be a string, while pt_handle should be of type `plc_pt_t`. The success or failure of this function is indicated by `pt_handle.valid`; non-zero means OK. This function, like `plc_init()`, should be called before the time-critical part begins.

`plc_update()`
: Updates the local (buffered) copy of all the MatPLC points. It should be called before reading MatPLC points or after writing them. If the module runs in a repeated "scan", this function will usually be called once at the beginning of the scan, and once at the end.

`value = plc_get(pt_handle)`
: Reads a MatPLC point (as it was at the time of the last `plc_update()`, or as changed by this module), returning it as an unsigned 32-bit integer. A 32-bit float version of this function is also available.

`plc_set(pt_handle, new_value)`
: Writes an unsigned 32-bit integer to a MatPLC point. A 32-bit float version of this function is also available. The write will not be seen by the other MatPLC modules until `plc_update()` is called.

The header file defines two types for work with these: `plc_pt_t`, which is used to declare the handles, and `u32` which is a simple 32-bit unsigned integer.

The two additional functions should be used if the module runs in a repeated scan, like a traditional PLC. They enable the MatPLC to enforce execution periods, sequencing of modules, RUN/STOP modes and the like. They take no arguments.

`plc_scan_beg()`
: Beginning of scan. This should precede the `plc_update()` call, as shown in the templates.

`plc_scan_end()`
: End of scan. This should follow the `plc_update()` call, as shown in the templates.

These functions are explained in more detail in the general reference [p 126] and GMM reference [p 129] chapters.

Other functions

Probably the most interesting ones will be the functions to retrieve values from the config. Unfortunately there's a lot of them, because they need to cover all the combinations of retrieving from single settings and tables, from the module's own section or from a different one, and converting the values to various types. They all start with `conffile_` and are described in detail in the conffile reference [p 135] .

As noted, there are floating-point equivalents to plc_get() and plc_set(); they are called plc_get_f32() and plc_set_f32(), work with the type f32 instead of u32, but otherwise behave exactly like the integer versions.

Notes

The MatPLC library is not particularly thread-safe. If you are using multiple threads, it would be best to restrict MatPLC functions to one thread. However, it is quite OK for one thread to be using the runtime functions while another thread does a `plc_pt_by_name()`.

[p 117] [p 1] [p 122]

$Date: 2005/05/15 09:37:00 $

[p 118] [p 1] [p 126]

Example custom module

This is the chaser module from the basic demo [p 20] .

At the top there's a couple of helper functions, `get_pt()` [p 123] and `get_delay()` [p 123] , for getting handles and the time delay from the MatPLC and checking that no error is returned.

After that we come to the `chase()` [p 123] function, which contains the bulk of the code. First it gets handles for the left and right buttons and all the lights, and then it gets the time delay value and starts up the timer. Then the main loop starts. Apart from the boiler-plate beginning and end of scan, it contains just two pieces of code: one to check the `left` and `right` buttons, and set the `dir` variable appropriately, and the other to move the light along one when the timer has expired.

The `main()` [p 124] function just initializes the library and calls `chase()`.

Listing

```
/*
 * (c) 2000 Jiri Baum
 *
 * Offered to the public under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2 of the
 * License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
 * Public License for more details.
 *
 * This code is made available on the understanding that it will not be
 * used in safety-critical situations without a full and competent review.
 */


/*
 * light chaser - sample code
 */

#include <stdio.h>
#include <math.h>

#include <plc.h>
#include <logic/timer.h>

const int num_lights = 4; /* how many lights in the chase, max 9 */

/*
 * Get a handle to a point, checking that it's valid. Abort if the handle
```

```
 * could not be obtained.
 */
plc_pt_t get_pt(const char *pt_name)
{
  plc_pt_t pt_handle;

  pt_handle = plc_pt_by_name(pt_name);

  if (!pt_handle.valid) {
    plc_log_errmsg(1, "Could not get handle to %s, aborting.", pt_name);
    printf("Could not get valid handle to %s.\n", pt_name);
    exit(1);
  }

  return pt_handle;
}

/* Obtain the chasing delay from the config, default 0.5 seconds. */
double get_delay(void)
{
  double res;

  /*
   * Get the value "delay", of type double, into res.
   * Minimum 0, maximum HUGE_VAL (ie, no maximum), default 0.5
   */
  if (conffile_get_value_d("delay", &res, 0, HUGE_VAL, 0.5) == 0) {
    /* a return value of 0 means OK - return the value obtained */
    return res;
  }

  /* some sort of problem - log it, and use half a second */
  plc_log_wrnmsg(1, "Could not get delay from config, using 0.5s");

  return 0.5;
}

void chase(void)
{
  plc_pt_t left, right, L[num_lights];
  char Lname[] = "Ln";
  int i, dir=1, cur=num_lights-1;
  double delay;
  plc_timer_t timer;

  /* get the point handles */
  left = get_pt("left");
  right = get_pt("right");
  for(i=0;i<num_lights;i++) {
    Lname[1] = '1'+i;
    L[i]=get_pt(Lname);
  }

  /* get the delay from the config */
  delay = get_delay();
  /* start the timer ticking */
  plc_timer_start(timer);
```

```
  /* now chase! */
  while (1) {
    /* beginning of scan */
    plc_scan_beg();
    plc_update();

    /* check for change of direction */
    if (plc_get(left) && !plc_get(right)) {
      dir = -1;
    } else if (plc_get(right)) {
      dir = 1;
    }

    /* has delay elapsed? */
    if (plc_timer_done(timer, delay)) {
      /* move the light along */
      plc_set(L[cur],0);
      cur=(cur+dir+num_lights) % num_lights; /* be careful about -1 % n */
      plc_set(L[cur],1);

      /* set timer back by delay; this avoids loss of precision */
      plc_timer_add(timer, -delay);
    }

    /* end of scan */
    plc_update();
    plc_scan_end();
  }
}

int main(int argc, char **argv)
{
  /* initialise the MatPLC library */
  if (plc_init("Chaser", argc, argv) < 0) {
    printf("Error initializing PLC\n");
    return -1;
  }

  chase();

  return 0;
}
```

Notes

If a module does just one thing, it does not need a timer - it can use the `scan_period` setting instead. For instance, the Data logger [p 89] works this way - it just writes the values to file or to the database as fast as it can, and the MatPLC library slows it down to the speed requested by the Application Builder. Most of the generic modules work this way.

The chaser needs a timer because it does two things at different time scales: it checks the left and right buttons as often as possible, but it only moves the light along a couple of times per second.

[p 118] [p 1] [p 126]

$Date: 2005/05/15 09:37:00 $

[p 122] [p 1] [p 129]

Custom modules - general functions reference

This section describes general functions and so on - functions that don't belong in any of the subsequent sections.

plc_init() **plc_done()**	`plc_init()` is used to initialize a private memory map for a module. This should be called once at the beginning of the module. `plc_done()` is used to shut down the library and private memory map for the module. This should be done if the module exits for any reason. Limitation: The `plc_done()` function is not particularly complete - it should shut down the module completely, deallocate everything and generally clean the slate enough so that `plc_init()` could be called again. This is not currently done, mostly through carelessness and because most of the examples are infinite loops, and the rest exit immediately after calling `plc_done()`. Please file a bug report or ask on the list if you need this fixed.	

	`plc_init(module_name, argc, argv);`
	
	`plc_done();`
		non real-time

<table>
<tr><td rowspan="3">plc_scan_beg()
plc_scan_end()</td><td colspan="2">These functions should be called at the beginning and end of each scan. Their effect is to signal the other modules and sleep until it is time for the next scan to begin, as configured by the <code>scan_period</code> and synch [p 46] options.

Normal usage of these functions is described in the basic custom module [p 118] chapter. In general terms, <code>plc_scan_beg()</code> should be the first thing in a scan, and <code>plc_scan_end()</code> the last.

Scans should not in general be nested; that is, make sure that these functions are called in alternation. For instance, if you have a long-running loop within your program, you might do the following in the middle:

<pre>plc_update() /* update outputs (may be omitted if nothing was plc_set()) */
plc_scan_end() /* end scan */
plc_scan_beg() /* begin new scan */
plc_update() /* update inputs */</pre>
If a module does not have discernible scans, these two functions can be omitted altogether. However, synch [p 46] , <code>scan_period</code> and similar functionality will not be available for modules that don’t call these functions.</td></tr>
<tr><td><pre>int plc_scan_beg(void);
int plc_scan_end(void);</pre></td><td>real-time; may sleep on semaphore or timer (as configured)</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td rowspan="2">plc_module_name()</td><td colspan="2">Returns the name of the module; memory must be de-allocated with free().</td></tr>
<tr><td><pre>const char *plc_module_name(void);</pre></td><td>non real-time</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td rowspan="2">plc_print_usage()</td><td colspan="2">Prints a usage summary fragment; call this from your own usage function to print out the command line options handled by plc_init().

Note: this is only applicable if you pass the real argc and argv to plc_init(). If you don’t pass argv in, it can’t do much about it...</td></tr>
<tr><td><pre>int plc_print_usage(FILE *output);</pre></td><td>non real-time</td></tr>
<tr><td colspan="3"></td></tr>
</table>

plc_init_try()	This function is internal and should not be used by general-purpose modules. FIXME: why is it public?	
	`int plc_init_try(char const *module_name, int argc, char **argv);`	non real-time
CLO_xxx definitions	Names of command-line options.	

	`#define CLO_LEADER "--PLC"`
	
	`#define CLO_plc_id CLO_LEADER "plc_id="`
	`#define CLO_loc_local CLO_LEADER "local"`
	`#define CLO_loc_isolate CLO_LEADER "isolate"`
	`#define CLO_loc_shared CLO_LEADER "shared"`
	`#define CLO_privmap_key CLO_LEADER "local_map_key="`
	`#define CLO_log_level CLO_LEADER "debug="`
	`#define CLO_log_file CLO_LEADER "logfile="`
	`#define CLO_config_file CLO_LEADER "conf="`
	`#define CLO_module_name CLO_LEADER "module="`
	`#define CLO_force_init CLO_LEADER "force-init"`
		n/a

[p 122] [p 1] [p 129]

$Date: 2005/05/16 07:08:36 $

[p 126] [p 1] [p 135]

Custom modules - gmm reference

The global map is the area where MatPLC points are kept.

It contains all of the application data, including physical I/O, internal coils, and any other data to be shared between modules. Private data may still be maintained inside the module, but be aware that if it is not in the global memory map, it is not going to be available to any other logic engines, HMI modules, debugging tools, etc.

Each module, upon being started, by default is given a private copy of the global memory map. All functions performed by a module are done to this private memory map which is then synchronized back to the global memory map through a function call in the shared memory library. This synchronization is semaphore controlled, which provides atomic updates. When run with a single logic engine, this allows the MatPLC to mimic the behaviour of a traditional PLC.

The shared memory is allocated when the MatPLC is started (`matplc -g`), and deallocated when the PLC is shut down (`matplc -s`). Even in the rare cases that MatPLC becomes inconsistent (e.g. while hacking the MatPLC kernel itself) the shutdown command usually manages to remove all the shared resources correctly. If for some reason some resources may reamin, these may be deallocated manually using `ipcrm(8)`.

The MatPLC gmm library is the only way to access the shared memory. At initialization, it allocates a private map and also a map mask. The mask is used to determine whether a module has write access to a particular data point. If a module tries to modify a data point to which it does not have write access, these changes will simply be ignored.

The library provides various functions for access to the global map, and some subsidiary functions, types and variables. Note the distinctions between private and global memory.

plc_pt_t	This is the "point handle" data type. The only field of `plc_pt_t` that is public is the `.valid` field. This indicates whether the point handle is properly initialized. Non-zero indicates success. Languages which have exceptions typically do not have a `.valid` field in the corresponding type. Point handles are usually obtained using the functions `plc_pt_by_name()`, `plc_subpt()` or `plc_pt_null()`.	
	`plc_pt_t data_point;`	

plc_get()	Provides read access to the private memory map. Note that this may not be an accurate representation of the data in the global memory map, as it is stored as a snapshot of data since the last call to `plc_update()`. If it is critical that the most accurate data be used, call `plc_update()` or `plc_update_pt(s)()` prior to `plc_get()`. There is no restriction to what data may be read in the private memory map.	
	`i=plc_get(data_point);` Data_point is of type plc_pt_t, and is checked to ensure it is a valid private memory map location. Please see the description of `plc_pt_t` above. The returned value is an unsigned integer of up to (currently) 32 bits, as read from the private map at the location addressed by data_point.	real-time; immediate
plc_set()	Provides write access to the private memory map. Note that when `plc_set()` is called, any changes to data not marked for write access in the shared memory manager's configuration will be discarded. Changes made to the private memory map will not be reflected in the global memory map until plc_update is called. If it is critical to get data changes to the global memory map, call `plc_update()` immediately after `plc_set()`.	
	`plc_set(data_point, value);` `data_point` is of type `plc_pt_t`, and is checked to ensure it is a valid private memory map location. Please see the description of `plc_pt_t` above. `value` is a 32-bit unsigned integer. This will be placed into `data_point` upon successful completion of the function.	real-time; immediate

<table>
<tr><td rowspan="2">plc_get_f32()
plc_set_f32()</td><td colspan="2">These have the same functionality as the <code>plc_get()</code> and <code>plc_set()</code> functions, except that the type of the value is a 32-bit float (<code>f32</code>). The value is simply stored as-is. It is up to the application to ensure that the point in question is also considered as a float by all other modules that access it.

The only value that is guaranteed to make sense when mixing types is that an integer zero will convert to a floating point zero. All others will result in nonsense values.</td></tr>
<tr><td><pre>value=plc_get(data_point);
plc_set(data_point, value);</pre>
<code>data_point</code> is of type <code>plc_pt_t</code>, and is checked to ensure it is a valid private memory map location. Please see the description of <code>plc_pt_t</code> above.

The returned or accepted <code>value</code> is a 32-bit float.</td><td>real-time; immediate</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td rowspan="2">plc_update()</td><td colspan="2">Causes the memory manager to re-synch the global memory map with the private memory map. Points to which the module calling <code>plc_update()</code> has right access are copied from the private to the global map. Read-only points will copied from the global map to the private map. The updates are semaphore controlled. This provides for atomic updates of the global memory map between all modules.

Some of the scripting languages, for instance Tcl [p 115] , do not explicitly have this function; instead, they call <code>plc_update()</code> every time a point is accessed.

Note: Partial updates of the global memory map are supported, but not documented here yet; see the <code>plc_update_pt()</code> and <code>plc_update_pts()</code> functions in <code>lib/gmm/gmm.h</code></td></tr>
<tr><td><pre>plc_update();</pre></td><td>real-time; may sleep on semaphore</td></tr>
<tr><td colspan="3"></td></tr>
</table>

<table>
<tr><td rowspan="2">plc_pt_by_name()</td><td colspan="2">Provides point handles of type <code>plc_pt_t</code> for use with the <code>plc_get()</code> and <code>plc_set()</code> functions. This function does not operate in real-time, as it must access the configuration file. All point handles should be established before time critical portions of the module start running.

In the object-oriented languages, this function is usually the constructor of the point class.

The only field of the <code>plc_pt_t</code> data type that is public is the <code>.valid</code> field. This indicates whether the point handle initialized properly or not. (Non-zero indicates success.)

Languages which have exceptions throw an exception when there is a problem; thus, there is no <code>.valid</code> field in those languages.</td></tr>
<tr><td><pre>data_point = plc_pt_by_name(data_point_name);
if (data_point.valid == 0) {
    printf("Error! Can't access %s \n",
           data_point_name);
}</pre></td><td>non-real-time</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td rowspan="2">plc_subpt()</td><td colspan="2">Creates sub-point handles of type plc_pt_t, which refer to a part of a 'parent' point. Sub-point handles can be used the same as normal point handles.</td></tr>
<tr><td>For example, given a 16-bit point, one can split it into two 8-bit points and access these separately.

<pre>/* get left half */
data_a_point = plc_subpt(data_point,0,8);
/* get right half */
data_b_point = plc_subpt(data_point,8,8);
if ((data_a_point.valid == 0) ||
    (data_b_point.valid == 0)) {
  printf("%s:%d: Internal error!\n",
         __FILE__,__LINE__);
}</pre></td><td>non-real-time</td></tr>
<tr><td colspan="3"></td></tr>
</table>

<table>
<tr><td rowspan="2">plc_pt_null()</td><td colspan="2">Creates a null point handle of type plc_pt_t. Null handles can be used the same as normal point handles. Anything written to them is discarded, on reading they return zero.

In object-oriented languages which allow polymorphic constructors, for instance python [p 113] , this function is usually a constructor of the point class with no arguments.</td></tr>
<tr><td><pre>
if (report_status) {
  status_point = plc_pt_by_name(status_point_name);
  if (status_point.valid == 0) {
      printf("Error! Can't access %s \n",
             status_point_name);
      status_point = plc_pt_null();
  }
} else {
  status_point = plc_pt_null();
}
</pre>
The rest of the program can then ignore the <code>report_status</code> flag and simply set and reset the <code>status_point</code> as appropriate.</td><td>non-real-time</td></tr>
</table>

Other gmm functions

There are other functions dealing with point handles. These are not documented here yet - see `lib/gmm/gmm.h` for more details

- variable modifying behaviour of plc_pt_by_name():
  - int plc_magic_bit_aliases;
- functions for obtaining handles:
  - plc_pt_by_index() /* get the n'th point */
  - plc_pt_by_loc() /* directly specify all details (dangerous) */
- number of configured points:
  - plc_pt_count() /* useful with plc_pt_by_index() */
- partial map updates:
  - plc_update_pt()
  - plc_update_pts()
- functions returning information about a handle:
  - plc_pt_len() /* length */
  - plc_pt_rw() /* read/write status */
  - plc_pt_rw_s() /* read/write status as a string */
  - plc_pt_details() /* low-level details */

[p 126] [p 1] [p 135]

$Date: 2005/05/15 09:37:00 $

[p 129] [p 1] [p 138]

Custom modules - conffile reference

The config file is the common mechanism for the user to specify settings and parameters. Pretty much everything should either be in there, or else in a file named by the config. It is strictly read-only; only the user may change the values in these files.

It is divided into sections: normally, each module will read data from its own section. However, it can also read other sections - for instance, every module looks in the `[PLC]` section for the semaphore keys and the list of points.

Retrieving values

There are numerous functions, to cover the various combinations of getting an ordinary value and a table value, default or explicit section, and conversions to numbers. The overall pattern of the function names is:

```
conffile_get_{value/table}[_sec][_type]
```

The arguments are as follows:

If there is an explicit section (`_sec`), it comes first.

The next argument (or first, if there's no section) is the name of the variable or table to be retrieved.

In the case of a table, the next two arguments are the row and the column numbers. (In some languages, the whole table will be returned as a two-dimensional array instead.)

If the *`type`* mark is omitted, the value is returned as a string, allocated with `malloc()`, or NULL if the value is not found (no such variable or table, or not that many rows or columns). For the other type marks (`_i32`, `_u32`, `_f32` and `_d`), the function takes a further four arguments: a pointer to the variable where the result is to be stored, a minimum, a maximum, and the default value to be used if the variable is not found. The function then returns a zero if the variable could be converted or if it was not found and the default is used, and non-zero if there was a problem.

In weakly typed languages, the *`type`* mark will not be available and all values will be returned as strings. If exceptions are available, they will be used to signal nonexistent values, otherwise empty strings will be returned (or as conventional for the language).

There is an additional set of functions for tables,

```
conffile_get_table_[rows/rowlen][_sec]
```

These return the number of rows in a table, or the length of a particular row. They return zero if there's no such table or row. Languages that return the whole table as a 2-d array will not have these functions.

Prototypes

Here are the prototypes for all of the above.

```
char *conffile_get_value(const char *name);
int conffile_get_value_i32(const char *name,
                           i32 *val, i32 min_val, i32 max_val, i32 def_val);
int conffile_get_value_u32(const char *name,
                           u32 *val, u32 min_val, u32 max_val, u32 def_val);
int conffile_get_value_f32(const char *name,
                           f32 *val, f32 min_val, f32 max_val, f32 def_val);
int conffile_get_value_d  (const char *name,
                           double *val, double min_val,
                           double max_val, double def_val);

char *conffile_get_value_sec(const char*section, const char *name);
int conffile_get_value_sec_i32(const char *section, const char *name,
                               i32 *val, i32 min_val, i32 max_val, i32 def_val);
int conffile_get_value_sec_u32(const char *section, const char *name,
                               u32 *val, u32 min_val, u32 max_val, u32 def_val);
int conffile_get_value_sec_f32(const char *section, const char *name,
                               f32 *val, f32 min_val, f32 max_val, f32 def_val);
int conffile_get_value_sec_d  (const char *section, const char *name,
                               double *val, double min_val,
                               double max_val, double def_val);

char *conffile_get_table(const char *name, int row, int col);
int conffile_get_table_i32(const char *name, int row, int col,
                           i32 *val, i32 min_val, i32 max_val, i32 def_val);
int conffile_get_table_u32(const char *name, int row, int col,
                           u32 *val, u32 min_val, u32 max_val, u32 def_val);
int conffile_get_table_f32(const char *name, int row, int col,
                           f32 *val, f32 min_val, f32 max_val, f32 def_val);

int conffile_get_table_rows(const char *name);
int conffile_get_table_rowlen(const char *name, int row);

char *conffile_get_table_sec(const char *section, const char *name,
                             int row, int col);
int conffile_get_table_sec_i32(const char *section, const char *name,
                               int row, int col,
                               i32 *val, i32 min_val, i32 max_val, i32 def_val);
int conffile_get_table_sec_u32(const char *section, const char *name,
                               int row, int col,
                               u32 *val, u32 min_val, u32 max_val, u32 def_val);
int conffile_get_table_sec_f32(const char *section, const char *name,
                               int row, int col,
                               f32 *val, f32 min_val, f32 max_val, f32 def_val);

int conffile_get_table_rows_sec(const char *section, const char *name);
int conffile_get_table_rowlen_sec(const char *section, const char *name,
                                  int row);
```

Other conffile functions

The functions `conffile_init()` and `conffile_done()` are called by the global `plc_init()` and `plc_done()`, so they generally shouldn't be called explicitly.

It's also possible to obtain the name of the *n*th variable in the config, using `conffile_var_by_index()`. For debugging, the config can be dumped with `conffile_dump()`.

I have no idea what the `conffile_parse_i32()` function is or how it differs from `conffile_get_value_i32()`.

[p 129] [p 1] [p 138]

$Date: 2004/12/28 05:32:09 $

[p 135] [p 1] [p 140]

Custom modules - log reference

The log is the common mechanism for errors, warnings and debugging information to be reported to the user.

There are three functions, one for each of the above, all with the same arguments. The first argument is the logging level, a number between 1 and 9 indicating the severity of the condition - see below. The remaining arguments are interpreted as the arguments to a `printf(3)`.

`void plc_log_trcmsg(int min_debug_level, const char *format, ...);`
: Log debugging information ("trace message"). This indicates successful execution of a specific action.

`void plc_log_wrnmsg(int min_debug_level, const char *format, ...);`
: Log a warning. Something has gone wrong, or unexpected result received, but an alternative/default/fallback action is available and will be tried (after this is logged). If all alternatives fail, only then log an error message.

`void plc_log_errmsg(int min_debug_level, const char *format, ...);`
: Log an error. Something is seriously wrong, for which no (further) corrective action is known, and the only solution is to give up on that specific action.

  Note that the PLC or the module that experienced the error does not necessarily have to stop executing (or crash), but may continue with other actions (e.g. modbus_tcp module cannot connect to a remote host 1 to read point A, but continues to connect to host 2 to get point B).

Logging levels

Logging levels are numbered from 1 to 9, with 1 the most important and 9 the least.

The default logging level is 1, which should be used by any part of the program to log messages that one would like to see by default. Typically, levels 2..5 will be used only by module specific code, while levels 6..9 will be used by the MatPLC libraries; however, it really depends on the severity of the event rather than its source.

Policy

The following table contains examples of the kinds of events that should be logged at each level.

	Debug	Warning	Error
1			
2			
3			
4			
5			
6			
7			
8			
9			

FIXME fill in the table.

All messages logged by the library or by generic and specific modules (that is, anything included with MatPLC itself) should be listed in the Log messages [p 164] appendix.

Real-time considerations

At present, this section of the library is not real-time; however, this functionality is planned - see SF ticket 766241

[p 135] [p 1] [p 140]

$Date: 2005/08/20 06:11:42 $

[p 138] [p 1] [p 142]

Custom modules - timer reference

The `timer` and `timer_ext` sections aren't part of the MatPLC library proper; if you want them, you have to explicitly link to them. They can be found in the `mat/lib/logic` directory.

timer

Timers are basically floats, with the additional feature that they can be enabled to increase at a rate of one per second. They are also fixed-precision, storing time as an exact number of microseconds.

Note that most of the functions listed below are actually macros; they modify the value of `timer` directly.

`plc_timer_t`
: The timer type.

`plc_timer_clear(timer)`
: Set to zero and disable. The timer is now zeroed and stopped.

`plc_timer_start(timer)`
: Set to zero and enable. The timer is now zeroed and running.

`plc_timer_enable(timer)`
: Enable the timer. This will make the value increase at a rate of one per second.

`plc_timer_disable(timer)`
: Disable the timer. This will make it keep whatever value it has at the moment without change.

`plc_timer_add(timer, value)`
: Add a number to the timer, effectively winding it forward. This works whether it's enabled or disabled. Use negative values to decrease the timer (wind it back).

`plc_timer_set(timer, value)`
: Set to a particular value. This works whether it's enabled or disabled. However, to avoid loss of precision, prefer to use `plc_timer_add()` whenever possible.

`double plc_timer_get(timer)`
: Get the current value of the timer. This works whether it's enabled or disabled.

`int plc_timer_done(timer, preset)`
: Check whether timer is at least `preset` seconds. This works whether it's enabled or disabled.

timer_ext

This is very similar to the above, with two differences: all the functions/macros end with `_ext`, and they all take an extra argument which they use as the current time instead of obtaining it from the operating system. The extra argument should be of type `struct timeval`, as used by the `gettimeofday(2)` system call.

One might use `timer_ext` if one runs in a conventional scan and doesn't want timers to expire mid-loop. To achieve this, call `gettimeofday(2)` once at the start of the loop and then use that time as the last argument to all the `timer_ext` calls.

Real-time considerations

FIXME

[p 138] [p 1] [p 142]

$Date: 2005/08/20 06:11:42 $

[p 140] [p 1] [p 146]

A custom I/O module

IO modules are all very similar, except for the way they access the hardware, and the format of how that hardware's IO addresses are specified in the configuration file. That is why Mario wrote an IO library to be used by IO modules. It essentially has everything required by an IO module (including the main() function), except for the functions to access the hardware and interpret the hardware's addresses specified in the config file.

Overview

What you need to do is write the functions listed below and link everything together - and presto, you have an IO module!

Some of the functions won't be relevant to your module; just write blank functions for those.

Initialization

Three functions to write here.

- `io_hw_parse_config()` - this function will be called first of all; if the module has any global options, this would be the place to read them from config.
- `io_hw_parse_io_addr()` - will be called once for each point. a function to translate a string (char *) into an address; if your inputs and outputs are numbered, you can just use `atoi()`; if it's more complicated, it'll depend on what it looks like.
- `io_hw_init()` - the rest of initialization - anything that needs to be done after all the addresses have been translated. If the module needs to open devices or similar, this should also be done here.

Input and output

- `io_hw_read()` - takes the address of an input, and reads it.
- `io_hw_write()` - takes the address of an output and a value, and writes the value to the output.

Miscellaneous

These functions are useful if you need them, but you probably don't. Most likely they'll be blank.

- `io_hw_read_end()` - called after each bunch of reads.
- `io_hw_write_end()` - called after each bunch of writes - useful if you need to flush a buffer.
- `io_hw_done()` - clean-up should go here, but since the module will terminate directly afterwards, it's not so important.

Details

You may assume that the `plc_init()`, `plc_update()`, `plc_scan_beg/end()` and `plc_done()` functions will be called for you at the appropriate times.

Your module will be given 8 bytes per I/O address to store the details; if this is insufficien, feel free to `malloc(3)` more space in the `io_hw_parse_config()` and/or `io_hw_parse_io_addr()` functions.

Order of calling

The I/O library will call the your functions in the following order:

- `io_hw_parse_config()`
- for each IO address in the map table
  - `io_hw_parse_io_addr()`
- `io_hw_init()`
- `while (1)`
  - for each input point
    - `io_hw_read()`
  - `io_hw_read_end()`
  - (note that `plc_update()` is called at this point)
  - for each output point
    - `io_hw_write()`
  - `io_hw_write_end()`
- `io_hw_done()`

Function prototypes

Here are the functions you need to define. (The actual definition is in the `lib/io/io_hw.h` file; if there's a clash, please let us know so we can fix this manual.)

`const char *IO_MODULE_DEFAULT_NAME`
: The default name of the IO module.

`int io_hw_parse_config(void)`
: This function is called first so the module gets a chance to parse any specific configuration before any other functions get called.

  For instance, it might parse a list of devices with device parameters. The individual I/O addresses would then use the names of these devices, as given in this list.

  It should also check that the size of whatever you're casting io_addr to is no more than the size of io_addr_t, like this:

```
if (sizeof(my_io_addr_t) > sizeof(io_addr_t)) {    plc_log_errmsg(1,"Datatype size problem.");    return -1;  }
```

Should return 0 on success, -1 on failure.

`int io_hw_parse_io_addr(io_addr_t *io_addr, const char *addr_stri, dir_t dir, int pt_len)`

This function must parse the I/O address described in the string `addr_stri`, and store it in `*io_addr`, which is 8 bytes long. Should return 0 on success, -1 on failure.

The direction and point-length values may be used for consistency checking. The definition of `dir_t is enum { dir_in, dir_out, dir_none }`

`int io_hw_init(void)`

Connect to hardware, etc. This function is called at the end of start-up, after all the I/O addresses have been parsed, so it can also do whatever housekeeping is required, allocate buffers depending on the number of I/O addresses used, and so on.

`int io_hw_write(io_addr_t *io_addr, u32 value)`
`int io_hw_read (io_addr_t *io_addr, u32 *value)`

These functions will be called to read/write a single point at a time. They should send/store the result in the 'value' variable

Should return 0 on success, -1 on failure.

`int io_hw_write_end(void)`
`int io_hw_read_end (void)`

These are useful when the hardware itself is double buffered - you may need to do some special stuff with the hardware to tell it to update the outputs with the values you have been writing, or similar. If nothing special needs to be done at this point, simply `return 0;`

Should return 0 on success, -1 on failure.

`int io_hw_done(void)`

Terminate connection to hardware and generally shut down. Should return 0 on success, -1 on failure.

`int io_hw_dump_config(int debug_level)`

Should dump to plc_log_trcmsg(debug_level, ...) any configuration parameters it will use. Used only for debugging purposes. Please use the debug_level specified.

Should return 0 on success, -1 on failure.

`char *io_hw_ioaddr2str(io_addr_t *io_addr)`

Should return a string description of the io_addr. Memory for the string must be `malloc()`'d, and will be `free()`'d by the calling function in the io library.

Overriding the scan loop

If the module is not suited to the standard scan cycle, as for instance many network and bus slaves aren't, this can be over-ridden by setting the `run_loop` callback.

Just write a function the way you want it, and in one of the setup functions, set run_loop to point to it, thus:

```
int my_run_loop(void*foo) {  ...}int io_hw_init(void) {  run_loop = my_run_loop;  ...}
```

The function `my_run_loop()` should return a negative value on error. It is not expected to return on success, but if it does, the value should be non-negative. The parameter is an internal structure, access to which has not been thought through at this point :-) We apologise for the inconvenience.

Utility functions

`plc_pt_t io_status_pt(const char *base, const char *suffix, int loglevel)`

This function looks for a point called *base.suffix*. If it exists, it's returned. If it doesn't exist, a warning is logged at `loglevel`, and a null point is returned.

The upshot is that if an optional status point is desired, one simply calls this function and uses the returned point as the status point. The user will take advantage of it, or not.

The `base` should be the name of the point to which the status point will refer. The `suffix` should indicate what kind of status is reported: it might be `"ok"`, `"err"`, `"timeout"`, `"errno"`, or similar, depending on what condition is to be indicated by the status point.

Slave mode

An IO module may work in one of two modes: master or slave. By default it will work in the master mode.

To run in slave mode, where it responds to requests from the bus... **FIXME**

[p 140] [p 1] [p 146]

$Date: 2005/08/20 06:11:42 $

[p 142] [p 1] [p 148]

Contributing to the MatPLC

There are three ways of contributing to the MatPLC, depending on your coding skill and the degree to which you are familiar (or wish to become familiar) with the MatPLC.

Testing

Testing, and reporting successes or failures, is the simplest way of contributing to the MatPLC. The developers do their own testing, of course, but sometimes we are a bit too close to the system to be able to test it well, and in any case there's only so much hardware we have access to. The only way to ensure robustness in a project such as this is for a large number of people to test it with a reasonable variety of hardware.

The documentation for this are the relevant sections of this manual, skipping perhaps the section on custom modules. Please send your reports to the mailing list, or submit bug reports to the SourceForge buglist. If you're reporting error messages from the MatPLC, please cut'n'paste - do not paraphrase.

Modules

Writing modules is the second way of contributing. It requires some skill in programming, and moderate knowledge of the MatPLC.

Writing modules to contribute is no different from writing custom modules [p 110] , so you'll need to read that section of the manual. Since it is currently rather sparse, please ask questions on the mailing list - we'll expand the manual as we answer them.

These modules can be either I/O modules, interfacing the MatPLC to particular hardware, or other modules, fulfilling some other useful function. Notice that I/O modules have an additional library available to them, which further reduces the amount of work required to write them.

Alternately, you can write **enhancements** (or even just bug fixes) for existing modules. This has the advantage that it is overall a smaller task, since the bulk of the module will remain unchanged. Take a look at the buglist on SourceForge if you would like to work on a module bugfix or enhancement; the wishlist on the other hand contains ideas for pure enhancements and new modules.

Core

Contributing to the MatPLC core requires familiarity with the internals of the MatPLC, and therefore an investment in time to learn them.

Read the whole manual, and any other documentation in the doc directory, in particular the article (doc/lplc-article.txt), and participate on the mailing list. Again, the Sourceforge buglist and wishlist contain lists of bugs to be fixed and new features that would be desirable, but it is by no means exhaustive. Discuss any proposed major changes on the mat-devel mailing list before beginning them - some of the developers may already be working on something similar.

[p 142] [p 1] [p 148]

$Date: 2004/12/28 05:32:10 $

[p 146] [p 1] [p 151]

Interfacing other projects to the MatPLC

Interfacing the MatPLC and another project will of course always depend on the details of the other project. However, typically, it would be done by making the other project (or some part of it) a MatPLC module. In some cases, an entirely new binary would be written, which is both a MatPLC module and interacts with the other project in some way. In either case, some part of the interface will be a MatPLC module.

The minimum required for MatPLC integration is to link with the shared (or static) library, and call five functions, with seven being preferable. The functions are:

`plc_init(module_name,0,0)`
: Initializes the library. The module_name should be a string.

`pt_handle = plc_pt_by_name(pt_name)`
: Obtains a handle to a MatPLC point, required for the `plc_get()` and `plc_set()` functions. The pt_name should be a string, while pt_handle should be of type `plc_pt_t`. This function, like `plc_init()`, should be called before the time-critical part begins.

`plc_update()`
: Updates the local (buffered) copy of all the MatPLC points. It should be called before reading MatPLC points or after writing them. If the module runs in a repeated "scan", this function will usually be called once at the beginning of the scan, and once at the end.

`value = plc_get(pt_handle)`
: Reads a MatPLC point (as it was at the time of the last `plc_update()`, or as changed by this module), returning it as an unsigned 32-bit integer. A 32-bit float version of this function is also available.

`plc_set(pt_handle, new_value)`
: Writes an unsigned 32-bit integer to a MatPLC point. A 32-bit float version of this function is also available. The write will not be seen by the other MatPLC modules until `plc_update()` is called.

The two additional functions should be used if the module runs in a repeated scan, like a traditional PLC. They enable the MatPLC to enforce execution periods, sequencing of modules, RUN/STOP modes and the like. They take no arguments.

`plc_scan_beg()`
: Beginning of scan. This should precede the `plc_update()` call.

`plc_scan_end()`
: End of scan. This should follow the `plc_update()` call.

Naturally, you’re welcome to use any other MatPLC functions that look useful.

These functions and their use are explained in more detail in the Custom modules [p 110] chapter, especially in the Native C [p 117] section.

Usual usage

There are two usual ways of structuring modules: the first is the "classical scan" familiar from stepladder, while the second is a "data interface", where MatPLC and some other system exchange data.

Classical scan

Here, the behaviour resembles a stepladder scan: at the top of the loop, data is read from the inputs (and other parts of the MatPLC), then the logic is executed, and then data is written to the outputs (and other parts of the MatPLC).

This is the template for a classical-scan module:

```
#include <plc.h>

int main(int argc,char *argv[])
{
  plc_init("modulename",argc,argv);

  /* initialization goes here */

  while (1) {
    plc_scan_beg();
    plc_update();

    /* body of loop goes here */

    plc_update();
    plc_scan_end();
  }
}
```

Data interface

Here, the module does data interchange between MatPLC and something else (the "other"). In the first half of the loop, any data that’s going from the other to the MatPLC is written into MatPLC points. In the second half of the loop, data is taken from MatPLC points and sent to the other. All the I/O modules work this way, exchanging data between MatPLC and the I/O.

One advantage is that the module doesn’t need to keep track of which points are read-only and which are read-write in the MatPLC; in the first half, all points may be written, and in the second half all points may be read. The `plc_update()` discards values for which the module lacks write permission.

This is the template for a data-interface module:

```
#include <plc.h>

int main(int argc,char *argv[])
{
  plc_init("modulename",argc,argv);

  /* initialization goes here */

  while (1) {
    plc_scan_beg();

    /* process data going to MatPLC */

    plc_update();

    /* process data coming from MatPLC */

    plc_scan_end();
  }
}
```

Notes

The MatPLC library is not particularly thread-safe. If you are using multiple threads, it would be best to restrict MatPLC functions to one thread. However, it is quite OK for one thread to be using the runtime functions while another thread does a `plc_pt_by_name()`.

[p 146] [p 1] [p 151]

$Date: 2004/12/28 05:32:10 $

[p 148] [p 1] [p 153]

Glossary

core
: The central part of the MatPLC; virtual backplane. Each module plugs into this core.

custom-made module
: A module written in the C language because none of the generic or specific modules provided by the MatPLC were suitable.

functional specification
: A written statement detailing what a system will do.

generic module
: A module that comes with the MatPLC and is used without modification.

HMI module (or MMI module)
: A module that interacts with the operator.

I/O module
: A module that connects to the "real world", or sometimes to a slave PLC over a bus or other connection.

Linux
: The operating system for which the MatPLC is designed.

`matplc.conf`
: The traditional name for the main configuration file of the MatPLC.

logic engine
: A module that does the actual logic, decision making and calculation.

modularity
: A quality of a system where it consists of various parts which separate cleanly and fit together well. High modularity costs some design time but pays back well through clarity, elegance, maintainability and flexibility.

module
: A component of the MatPLC architecture. Several modules running together make a useful MatPLC. Note that other parts of an installation may also have modules - the machine itself, I/O hardware, the software you write for the MatPLC, the Linux kernel, etc.

point
: The basic unit of storage and communication within the MatPLC. Points may be 1-bit, corresponding to discrete inputs, outputs, internal/memory coils, flags &c, or multi-bit (up to 32-bit), corresponding to analog inputs, outputs, integer or floating point registers &c. Each point has a name by which it is known throughout the MatPLC.

scan
: Many modules, especially logic engines, work in a cycle. One repeat is called a scan. It begins by obtaining a copy of all the points from the MatPLC core. It then evaluates all the logic or does all the calculations (depending on the module). Finally, all changed points are written back to the MatPLC core - this only happens at the end of the scan, so if the logic wrote a point several times with different values, only the last value is ever seen by any other module.

specific module
: A module that is created for a particular project using the tools provided by the MatPLC. Unlike a generic module, it does not directly come with the MatPLC.

[p 148] [p 1] [p 153]

$Date: 2004/12/28 05:32:09 $

[p 151] [p 1] [p 164]

Appendix: matplc.conf

```
#
# sample matplc.conf configuration file
#



#########################################################
#                                                       #
#     C O N F I G U R A T I O N     S Y N T A X         #
#     ========================================          #
#                                                       #
#########################################################




#   G E N E R A L   S Y N T A X
#   ===========================
#
# (name,value) pairs:
# name = <value>
#
# tables:
# table_name <value_1_1> <value_1_2> <value_1_3> ...
# table_name <value_2_1> <value_2_2> <value_2_3> ...
# ...
#
#
# tables and names are divided into sections.
# (Jiri, do you think you can come up with some good sentences
#  explainig how sections work, and how modules look for parameters
#  in sections named after their module_name ??)
# Should also explain how #includes are supported, and that it is protected
# against circular includes.
#
#
# NOTES:
#  (1)  when value is an integer, it may usually be written in
#       hexadecimal (0xF45A), octal (0172132) or decimal (62554) formats.
#
#  (2)  when value or a name is a string with white space characters, it
#       must be enclosed within "  ("this is a single value")
```

```
[PLC]
#
#   T h e   P L C   s e c t i o n
#   ============================
#
#
# The PLC section is used to configure parameters related to the PLC, and
# are used by all modules. Some other parameters related to the PLC, but
# module specific, must be specified under those modules' sections.
#
#
# Note that the synchronisation system uses a petri net to synchronise the
# module execution. This petri net may be configured by _one_ of two methods.
# The first, the simplified method, is merely the definition of the sequence
# by which the modules should execute. This method is configured with the
# synch and synch_start tables.
# The second, more complete and complicated method, requires the complete
# definition of the petri net. Although more complex, this method is also
# more powerfull, and supports some synchronisation sequences that can not be
# configured by the simple method. This complex method is configured by the
# place, transition, arc, and synchpt tables, and the synchpt_beg/end values.
#
#
#
# name = value pairs:
# -------------------
# max_modules   = <x> : The maximum number of simultaneously active modules
#                       the MatPLC will ever have.
#                       If this parameter is not specified, the
#                       default value STATE_MMC_DEF (currenty = 20) is used.
#                       Valid values go from 0..MMC_MAX. MMC_MAX depends on
#                       the operating system currently being used, and the
#                       number of places in the synchronisation Petri Net.
#                       Please read on for an explanation on the synchronisation
#                       Petri Net.
#
# confmap_key   = <x> : The key to use for the confmap shared memory. This is
#                       actually the same parameter configured through
#                       --PLCplc_id=xxx. The command line argument takes
#                       precedence over the value configured in this file.
#                       If this parameter is not specified either way, the
#                       default value DEF_CONFMAP_KEY (currenty = 23) is used.
#                       Valid values go from 0..INT_MAX. If 0 is specified,
#                       then a random key is chosen.
#
# confsem_key   = <x> : The id of the sempahore set used by the CMM.
#                       If no value is specified, then the default
#                       GMM_DEF_SEM_KEY is used (currently = 0). Valid values
#                       go from 0..INT_MAX. If 0 is specified, then a random
#                       key is chosen.
#
# confmap_pg    = <x> : The size of the confmap given in number of memory pages.
#                       If this value is left unspecified, the default value
#                       DEF_CONFMAP_PG (currently = 2) is used.
#                       The size of each memory page is given by PAGE_SIZE
#                       defined in asm/page.h or sys/user.h (which is the
#                       correct include?) (for linux on intel (?) PAGE_SIZE
#                       is 4 kBytes).
```

```
#
# globalmap_key = <x> : The key to use for the globalmap shared memory segment.
#                       If no value is specified, then the default
#                       DEF_GLOBALMAP_KEY is used (currently = 0). Valid values
#                       go from 0..INT_MAX. If 0 is specified, then a random
#                       key is chosen.
#
# globalsem_key = <x> : The key to use for the globalmap semaphore set used by
#                       the GMM. If no value is specified, then the default
#                       DEF_SEM_KEY is used (currently = 0). Valid values
#                       go from 0..INT_MAX. If 0 is specified, then a random
#                       key is chosen.
#
# globalmap_pg  = <x> : The size of the globalmap given in number of memory
#                       pages. If this value is left unspecified, the default
#                       value DEF_CONFMAP_PG (currently = 2) is used.
#
# synchsem_key  = <x> : The id of the sempahore set used by the SYNCH sub-system.
#                       If no value is specified, then the default
#                       DEF_SYNCH_SEM_KEY is used (currently = 0). Valid values
#                       go from 0..INT_MAX. If 0 is specified, then a random
#                       key is chosen.
#
#
# tables:
# ------
# module     : defines which modules should be started by the plc setup program.
# point      : defines the named points in the plc. See further down for syntax.
# point_alias: defines aliases to the named points in the plc. See further
#              down for syntax.
# synch      : used by the simple method of configuring module synchronisation.
#              It specifies the module depencies and how they should synchronise
#              between themselves.
# synch_start: used by the simple method of configuring module synchronisation.
#              It specifies which modules should be allowed to execute the first scan
#              as soon as the plc is started.
# place      : used by the complex method of configuring module synchronisation.
#              It specifies the places in the synchronisation Petri net.
# transition : used by the complex method of configuring module synchronisation.
#              It specifies the transitions in the synchronisation Petri net.
# arc        : used by the complex method of configuring module synchronisation.
#              It specifies the arcs in the synchronisation Petri net.
# synchpt    : used by the complex method of configuring module synchronisation.
#              It specifies the synchpts in the synchronisation Petri net.
#
#
#
# the MODULE table
# ----------------
# syntax:
# module <name> <file> [<options> ...]
#
# where:
# name      = the name of the module. This name will be used in sections, module
#             synchronisation, etc...
# file      = the program that will be executed.
# options   = other command line options that need to pe passed to the program.
#
#
#
```

```
# The POINT table
# ---------------
# syntax:
# point <name> <full name> <owner> [at <offset>[.<bit>]] [[i|u|f]<length>] [init <init_val>]
#
# where:
# point     = 'point' identifier
# name      = Name used to search for point in plc_pt_by_name()
# full name = More extensive description of the point
#             This information is not loaded into the confmap of the plc
# owner     = Name of the module with write permission on the point
# at        = 'at' identifier
#             This, along with the <offset>.<bit> is optional. When not
#             explicitly specified, the plc will try to find an empty slot
#             for the point.
# offset    = location of word in the globalmap that holds the point's state.
# bit       = the bit, in the word, that holds the point's state. This is
#             optional, defaulting to 0.
# i | u | f = the 'i', 'u' or 'f' identifiers.
#             These are used to specify the format of the data in the plc point.
#             At the moment this is only used by the plc code to figure out
#             how to interpret the initial value of the point when parsing
#             the init_val parameter.
# length    = the size in bits of the point. The point must not overflow
#             onto the next offset position (i.e. (bit + length) <
#             (8*sizeof (u32)).
#             The length is optional, defaulting to:
#                1 if the {at <offset>.<bit>} is not explicitly specified.
#               32 if {at <offset>} is specified without explicitly
#                  specifying the <bit>
#                1 if both {at <offset>.<bit>} are explicitly specified.
# init      = 'init' identifier
# init_val  = the initial value of the plc point. Defaults to 0.
#             If it is not specifically specified how to interpret the init_val
#             using the [i|u|f] identifiers before the length of the point,
#             then it will be interpreted as a 32 bit float (f32)
#             if it contains a floating point (e.g. 1.1) or an exponent (e.g. 1e9)
#             If the above is not true, then the init_val will be interpreted
#             as an xx bit signed integer if it has a leading '+' or '-' character.
#             If none of the above apply, then the value will be interpreted as
#             an xx bit unsigned integer.
#             The xx is the length of the plc point. Only 32 bit floats, integers
#             or unsigned integers are currently supported.
#
#
#
# the POINT_ALIAS table
# ---------------------
# syntax:
# point_alias <name> <full name> <org_name> [<bit> [<length>]]
#
# where:
# point_alias = 'point_alias' identifier
# name        = Name used to search for point in plc_pt_by_name()
# full name = More extensive description of the point
#             This information is not loaded into the confmap of the plc
# bit       = The first bit, of the original point, that this alias will
#             reference. This is optional, defaulting to 0.
# length    = the size in bits of the point. The point must not overflow
#             outside the original point (i.e. (bit + length) <
```

```
#             (length of org_name point). The length is optional, defaulting
#             to the length of the org_name point if no bit is explicitly
#             specified, and 1 otherwise.
#
#
#
# the SYNCH table
# ---------------
# syntax:
# synch <module_1> arrow <module_2>
#
# where:
# module_X = The name of a module. Whenever module_1 finishes executing it's scan,
#            then module_2 will start executing a new scan.
# arrow    = is an arrow with the same syntax as the one specified for the arrow
#            in the arc table, but with the additional constraint that weights
#            with a value of 0 are not valid for this table.
#
#
#
# the SYNCH_START table
# ---------------------
# syntax:
# synch_start <module_1> [<module_2> ...]
#
# where:
# module_X = The name of a module. Identifies which modules should be allowed to start
#            executing their first scan without any synchronisations. In other words,
#            which modules should start executing imediately once the PLC is started.
#
#
#
# the PLACE table
# ---------------
# syntax:
# place <name> [<init_tokens>]
#
# where:
# name        = The name of the place. The names specified on this table will later be
#               used in the arc table.
# init_tokens = The number of tokens the place should have when the PLC is started. If
#               not specified, it defaults to 0.
#
#
#
# the TRANSITION table
# --------------------
# syntax:
# transition <name>
#
# where:
# name        = The name of the transition. The names specified on this table will
#               later be used in the arc and synchpt tables.
#
# NOTE: The transition name "NULL" is reserved for the null transition.
#       Wherever a name = value pair requires the name of a transition for the
#       value, the name "NULL" may be used to specify the null transition.
#
#
# the ARC table
```

```
# -------------
# syntax:
# arc <place> arrow <transition>
#   or
# arc <transition> arrow <place>
#
# where:
# place      = The name of a place specified in the place table.
# transition = The name of a transition specified in the transition table.
# arrow      = is an arrow with the following syntax:
#                [-][(][<weight>][)][-]>
#              where weight is a non negative integer.
#              Possible correct examples are:
#                ->         # weight = 1
#                ----->     # weight = 1
#                >          # weight = 1
#                -()->      # weight = 1
#                ()>        # weight = 1
#                (>         # weight = 1
#                )>         # weight = 1
#                -7->       # weight = 7
#                8>         # weight = 8
#                (9)>       # weight = 9
#                (9>        # weight = 9
#                9)>        # weight = 9
#                etc...
#              Even though all above examples are correct, we sugest using:
#                ->               : for arrows with weight = 1
#                -X->  or  -(X)->  : for arrows with weight = X
#
# NOTE: arcs with weight 0 are only allowed from <place> --> <transition>









[module_name]
#
#  P L C   r e l a t e d   v a l u e s
#  ===================================
#
#  PLC related values, specific to each module.
#
# name = value pairs:
# ------------------
# location = <x> : defines how the module should access the shared plc
#                  resources. Possible values are "shared", "local" and
#                  "isolate".
#                  This is the same as the command line --PLClocal,
#                  --PLCisolate and --PLCshared arguments. The command line
#                  arguments take precedence over whatever is defined in the
#                  config file. If no value is specified by either method,
```

```
#                  the default DEF_PLC_LOCATION (currently = local) is used.
#                  Detailed explanation:
#                  "shared" provides direct access to the shared plc
#                  resources, while both "local" and "isolate" use a local
#                  copy which is synchronised once every scan cycle.
#                  The "local" version acesses the shared resources directly
#                  when synchronising to the shared resources, while
#                  "isolate" asks a proxy process to do the synchronisation.
#                  The "shared" version has the advantage of not requiring
#                  to synchronise the local copy to the shared resources,
#                  which may be somewhat time comsuming. Unfortunately,
#                  and precisely because it does not use any synchronisation
#                  mechanisms, it may produce incorrect results if more
#                  than one module tries to access the shared resources
#                  simultaneously. This method should only be used if the
#                  modules are guaranteed not to run simultaneously, for
#                  example, by correctly configuring the inter-module
#                  synchronisation mechanism ("synch" table, etc...)
#                  The "local" and "isolate" methods do not have the above
#                  limitations since they keeps a local copy of the shared
#                  resources, which may be safely accessed any time by the
#                  module. The local copy is then typically synchronised with
#                  the shared resources once every scan cycle. Since the
#                  synchronisation is protected by synchronisation mechanisms
#                  to gurantee that only one module may access the shared
#                  resources at a time, these methods may be used safely,
#                  whatever the execution sequence of the modules may be.
#                  The "isolate" and "local" methods merely differ on who
#                  does the actual synchronisation between the local copy and
#                  the shared resources. In the "local" method the module
#                  itself accesses the shared resources while doing the
#                  the synchronisation. In the "isolate" method, the
#                  module asks a proxy process to do the synchronisation.
#                  Note that the proxy process is launched automagically.
#                  The "local" method, although faster, has the drawback that
#                  a bug in the module code may result in the shared
#                  resources being changed involuntarily (e.g. by a stray
#                  pointer). For this reason, we recomend that the "isolate"
#                  method be used when a module is being tested, and the "local"
#                  method be used during normal runs.
#
# privatemap_key = <x>: defines the key to use for the local map shared memory.
#                       Valid values go from 0..INT_MAX. A value of 0 has
#                       diferent meanings, depending on the location value.
#                       For local access, a value of 0 implies the use of malloc
#                       to allocate memry for the local map. For isolate access,
#                       a random key is used for the local map shared memory.
#                       The default value for every access method is
#                       DEF_LOCALMAP_KEY (currently = 0)
#                       NOTE: in the source code the localmap is also sometimes
#                       refered to as the private map. This is something that
#                       must be corrected...
#
# scan_period = <x> : defines the scan period, in seconds, of the module. The PLC
#                     interprets values down to 1 ns, but the precision is limited
#                     to the resolution of the clock of the hardware on which it is
#                     running. On plain vanilla linux on x86 hardware the resolution
```

```
#                    is normally 10 ms.
#                    Note that the effective scan period of a module
#                    also depends on how it synchronises with every other
#                    module of the PLC. When a module finishes executing it's scan,
#                    it will first wait until it's scan period has elapsed, and then
#                    synchronise with the remaining modules. Only then will it
#                    actually execute the next scan. This means that the effective
#                    scan period may end up being larger than the one specified in
#                    this parameter.
#                    The scan periods are counted on an absolute time frame. Consider
#                    t0 the instant the first scan started. The following scans will
#                    be enabled at t0+T, to+2*T, t0+3*T, etc...
#
# scan_beg = <x> : used by the complex method of specifying the synchronisation
#                  dependencies.
#                  Identifies the transition to which the module should synch at the
#                  begining of every scan. If no transition is specified, the default
#                  NULL transition is used. Note that synchronising to NULL transition
#                  always returns without any delay, so by default the module will not
#                  synchronise to anything.
#
# scan_end = <x> : used by the complex method of specifying the synchronisation
#                  dependencies.
#                  Identifies the transition to which the module should synch at the
#                  end of every scan. If no transition is specified, the default
#                  NULL transition is used. Note that synchronising to NULL transition
#                  always returns without any delay, so by default the module will not
#                  synchronise to anything.
#
```

```
#############################################################
#                                                           #
#     S A M P L E     C O N F I G U R A T I O N S           #
#     ===========================================           #
#                                                           #
#############################################################
```

```
#
#  C O N F I G   (1) : Absolute Minimum Configuration
#  ================================================
#
```

```
[PLC]
# we need some points, otherwise this is a useless exercise...
point P0.0  "full name 0.0" module1 at  0.0      # 1 bit point at 0.0
point P0.1  "full name 0.1" module1 at  0.1  5   # 5 bit point from 0.1 to 0.5
point P1    "full name 1.0" module2 at  1        # 32 bit point from 1.0 to 1.31

[PLC]
# We also need some module to execute...
module module1 /matplc.logic/...

[module1]
# Here we would insert module specific configurations




#
#  C O N F I G   (2) : Configuration with Simple Synchronisation
#  ============================================================
#

[PLC]
# we need some points, otherwise this is a useless exercise...
point P0.0  "full name 0.0" module1 at  0.0      # 1 bit point at 0.0
point P0.1  "full name 0.1" module1 at  0.1  5   # 5 bit point from 0.1 to 0.5
point P1    "full name 1.0" module2 at  1        # 32 bit point from 1.0 to 1.31

[PLC]
# We also need some module to execute...
module module1 /matplc.logic/...
module module2 /matplc.logic/...

[PLC]
# We specify that the modules must run synchronously, one after the other
#
# After module1, run module2, and vice-versa
synch module1 -> module2
synch module2 -> module1

# start off by running module1
synch_start module1

# In addition, if we wanted the scans to execute only every 0.1 seconds, we can
# specify the scan period of one of the modules.
# Since both modules run in lock-step, we must only specify the scan period of *one*
# of the modules...
module1: scan_period = 0.1



[module1]
# Here we would insert module specific configurations
```

```
[module2]
# Here we would insert module specific configurations




#
#  C O N F I G   (3) : Configuration with Complex Synchronisation
#  ================================================================
#

[PLC]
# we need some points, otherwise this is a useless exercise...
point P0.0  "full name 0.0" module1 at  0.0      # 1 bit point at 0.0
point P0.1  "full name 0.1" module1 at  0.1  5   # 5 bit point from 0.1 to 0.5
point P1    "full name 1.0" module2 at  1        # 32 bit point from 1.0 to 1.31

[PLC]
# We also need some module to execute...
module module1 /matplc.logic/...
module module2 /matplc.logic/...



[PLC]
#
#  The synchronisation config
#  --------------------------
# Each module will use one place (P_module1, P_module2),
# and two transitions (T_module1_beg, T_module1_end, etc...).
#
# The number of tokens in P_module1 indicates the number of times
# module1 must run its scan loop. Likewise for module 2.
# At the beginning of the module1 scan this module waits on the T_module1_beg
# transition. This transition will remove one token from P_module1,
# and let the module start executing.
# At the end of the scan module1 will wait on the T_module1_end
# transition. This transition will place a token in P_module2,
# therfore allowing module2 to run.
#
# Module2 works in exactly the same way as module1.

place P_module1  1 # initialise with 1 token => start off by running this module.
place P_module2    # 0 is not required, it is the default value.

transition  T_module1_beg
transition  T_module1_end
transition  T_module2_beg
transition  T_module2_end

arc P_module1     -> T_module1_beg
arc T_module1_end -> P_module2
```

```
arc P_module2     -> T_module2_beg
arc T_module2_end -> P_module1

# The only thing left is to tell each module what transitions to use for the beg
# and end of the scan.
module1: scan_beg = T_module1_beg
module1: scan_end = T_module1_end
module2: scan_beg = T_module2_beg
module2: scan_end = T_module2_end



# In addition, if we wanted the scans to execute only every 0.1 seconds, we can
# specify the scan period of one of the modules.
# Since both modules run in lock-step, we must only specify the scan period of *one*
# of the modules...
module1: scan_period = 0.1
```

[p 151] [p 1] [p 164]

[p 153] [p 1] [p 165]

Log messages

This appendix lists the messages that may appear in the `matplc.log` file. It is intended to be complete (but isn't - yet), so if you get a message which is not listed here from one of the generic or specific modules, it's a problem - please let us know so we can fix it.

Messages are organised by module; with messages that may be produced by any module or by the MatPLC tools listed first in the "General" section.

General

type	text	explanation
TRACE	Module started.	The module started successfully.
TRACE	Launched 3 modules.	The `matplc -g` command has successfully started all the modules listed in `matplc.conf`. (This will always be labelled as coming from the "`plc_setup`" module.)
WARNING	The clock's resolution on this system (0.010000 sec) is not small enough to guarantee correct periodic execution of the module at the required period (0.010000 sec).	The timing of the module is likely to jitter badly; otherwise, everything's fine.
ERROR	Error initializing access to the plc config memory...	The `matplc -s` command couldn't find the MatPLC - most often this will happen when you do a `matplc -s` "just to make sure" and there isn't actually one running. (This will always be labelled as coming from the "`plc_shutdown`" module.)

[p 153] [p 1] [p 165]

$Date: 2004/12/28 05:32:09 $

[p 164] [p 1] [p 1]

MatPLC License terms

Unless otherwise specified in an individual module or other component, the MatPLC is licensed to the public under the terms of the GNU General Public License. The individual modules and pieces are Copyright © the various authors, as noted in the individual source files and other places. In the absence of a formal notice, the files are generally nevertheless Copyright © by their respective authors.

The GNU General Public License is reproduced below, or it can be obtained from the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. The Free Software Foundation also maintains a FAQ about the license, clarifying various points and issues.

Table of Contents

- GNU GENERAL PUBLIC LICENSE [p 165]
  - Preamble [p 165]
  - TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION [p 166]
  - How to Apply These Terms to Your New Programs [p 170]

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

```
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA  02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone’s free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with

the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- **a)** You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

- **b)** You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

- **c)** If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- **a)** Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

- **b)** Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete

machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

- **c)** Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES

ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) yyyy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.
```

The hypothetical commands `'show w'` and `'show c'` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `'show w'` and `'show c'`; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program ‘Gnomovision’
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

[p 164] [p 1] [p 1]